PERANCANGAN DAN PEMBUATAN PERANGKAT LUNAK PROTOTIPE BASIS DATA TERDISTRIBUSI DENGAN SISTEM CLIENT/SERVER

Disusun oleh :

JUNUS JUNARTO DJUNAWIDJAJA
NRP. 2693100001

JURUSAN TEKNIK INFORMATIKA
FAKULTAS TEKNOLOGI INDUSTRI
INSTITUT TEKNOLOGI SEPULUH NOPEMBER
SURABAYA
1998

PERANCANGAN DAN PEMBUATAN PERANGKAT LUNAK PROTOTIPE BASIS DATA TERDISTRIBUSI DENGAN SISTEM CLIENT/SERVER

TUGAS AKHIR

Diajukan Guna Memenuhi Sebagian Persyaratan
Untuk Memperoleh Gelar Sarjana Teknik Informatika
Pada
Jurusan Teknik Informatika
Fakultas Teknologi Industri
Institut Teknologi Sepuluh Nopember
S u r a b a y a

Mengetahui / Menyetujui

Dosen Pembimbing I	Dosen Pembimbing II
Dr. Ir. Arif Djunaidy	Ir. Muchammad Husni
NIP. 131 633 403	NIP. 131 411 100

SURABAYA
1998

Janganlah kamu menjadi serupa dengan dunia ini,
tetapi berubahlah oleh pembaharuan budimu,...
Roma 12:2

ABSTRAK

Pada saat ini, sebagian besar organisasi menggunakan sistem manajemen basis data tersentralisasi, dimana cabang-cabang organisasi semuanya berhubungan langsung ke sistem basis data yang di tempatkan di pusat. Kelemahan utama sistem basis data terpusat adalah jika terjadi kegagalan sistem basis data menyebabkan lumpuhnya keseluruhan sistem dan mahalnya biaya komunikasi data antara cabang-cabang dengan pusat organisasi. Pada sistem basis data terdistribusi, letak data dapat diimplementasikan secara tersebar, sehingga data tertentu yang paling sering diakses oleh bagian tertentu dapat dipecah ke lokasi-lokasi yang paling sering membutuhkannya, tanpa mengurangi kemampuan untuk mengakses data secara global.

Dalam Tugas Akhir ini, dibuat suatu prototipe sistem manajemen basis data terdistribusi. Sistem tersebut menggunakan dan menggabungkan barbagai macam sistem manajemen basis data (*Data Base Manajemen System* / DBMS) yang ada, sehingga secara global tampak bagi pemakai suatu sistem basis data terintegerasi yang utuh. Sebagai antar-muka untuk mengakses sistem ini dibuat sebuah *driver* yang mengacu pada standar JDBC (*Java DataBase Connectivity*). Prototipe sistem manajemen basis data terdistribusi ini menerapkan beberapa prinsip-prinsip dasar pada sistem manajemen basis data terdistribusi seperti masalah keamanan sistem dimana setiap pemakai yang akan mengakses data harus mengidentifikasi diri dan memasukkan *password*, pemecahan (fragmentasi) basis data dilakukan secara horisontal, aturan *semi-join* digunakan untuk menghemat beban komunikasi data, penggunaan metode *time-stamp* untuk masalah *concurrency control* agar dua atau lebih transaksi dapat berjalan bersama-sama secara *serializable*, serta dilakukannya replikasi untuk setiap data yang ada.

Manfaat penggunaan standar *driver* JDBC pada sistem ini adalah adanya kemudahan untuk menggantinya dengan *driver* JDBC milik DBMS lain, tanpa perlu melakukan kompilasi ulang dari program aplikasi. Keseluruhan pembuatan perangkat lunak prototipe sistem basis data terdistribusi ini menggunakan bahasa Java, sehingga memudahkannya untuk dijalankan di berbagai *platform* yang berbeda perangkat keras maupun sistem operasi.

KATA PENGANTAR

Segala puji syukur dan hormat penulis panjatkan kepada Allah Tri Tunggal sebagai sumber dari segala hikmat, karena segala anugrah dan kekuatanNya Tugas Akhir yang berjudul "**PERANCANGAN DAN PEMBUATAN PERANGKAT LUNAK PROTOTIPE BASIS DATA TERDISTRIBUSI DENGAN SISTEM CLIENT/SERVER**" ini dapat diselesaikan. Penulisan Tugas Akhir ini disusun guna memenuhi salah satu persyaratan untuk meraih gelar sarjana pada Jurusan Teknik Informatika Fakultas Teknologi Industri Institut Teknologi Sepuluh Nopember Surabaya.

Banyak bantuan yang penulis terima dalam penyelesaian tugas akhir ini kepada:

1. Bapak Ir. Arif Djunaidy, M.Sc, Ph.D, selaku dosen pembimbing dan Ketua Jurusan Teknik Informatika-ITS yang telah memberikan bimbingan dan motivasi.
2. Bapak Ir. Muchammad Husni, selaku dosen pembimbing yang telah memberikan bimbingan, bantuan dan motivasi.
3. Bapak Ir. Supeno Djanali, M.Sc, Ph.D, selaku dosen wali selama perkuliahan di Jurusan Teknik Informatika-ITS.
4. Bapak dan Ibu Dosen di Jurusan Teknik Informatika-ITS.
5. Seluruh staf tata usaha Jurusan Teknik Informatika-ITS.
6. Seluruh keluarga di Semarang, Mama, kakak Evi, dan adik Vera atas doa, motivasi, dan nasehatnya.
7. Daniel Oranova, atas segala dukungan dan bantuannya selama pengerjaan Tugas Akhir ini, khususnya peminjaman memori saat testing program dan demo.
8. Seluruh anggota rumah kontrakan di Mulyosari Tengah 9 / 17 atas bantuan dan motivasi baik yang serius maupun yang tidak. Diory, Sonny, Rony, serta Darwan.
9. Seluruh warga angkatan 93 Teknik Informatika-ITS yang telah membantu dan memotivasi penulis, teman-teman di lab Komisi, yang memberikan semangat dan

fasilitas komputer dan internet untuk mendownload program-program yang diperlukan, serta Lailil atas peminjaman printer.

10. Teman-teman di TPKK-ITS maupun di gereja yang telah memberikan dukungan doa dan motivasinya.
11. IPTEKNet ITS yang telah menyediakan fasilitas dan kesempatan bagi penulis untuk mempelajari jaringan komputer.
12. Semua pihak yang telah ikut membantu penulis selama menjalani studi dan tinggal di Surabaya.

Salah satu ciri karya manusia adalah tidak akan pernah sempurna karena manusia memiliki kelebihan dan kekurangan masing-masing. Demikian pula halnya dengan tugas akhir ini masih banyak kekurangannya. Untuk itu kritik dan saran yang bermanfaat sangat diharapkan demi penyempurnaannya. Harapan penulis juga, semoga tulisan ini bermanfaat bagi pembaca.

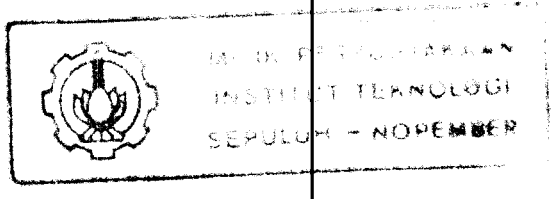
Surabaya, Juni 1998

Penulis

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL



BAB I

PENDAHULUAN

Dalam bab ini dijelaskan beberapa hal dasar yang meliputi latar belakang, permasalahan, tujuan, batasan permasalahan, metodologi serta sistematika pembahasan buku tugas akhir ini. Dari uraian tersebut diharapkan, gambaran umum permasalahan dan pemecahan yang diambil dapat dipahami dengan baik.

1.1. Latar Belakang

Peranan informasi semakin penting dalam era sekarang ini, dimana informasi menjadi elemen terpenting dalam proses perencanaan dan pengambilan keputusan. Pengolahan sistem informasi berbasis komputer biasanya menggunakan suatu Sistem Manajemen Basis Data (*Data Base Manajemen System*/DBMS). Pada saat ini sistem basis data yang biasa digunakan adalah sistem basis data terpusat (sentralisasi), padahal secara fisik data itu tersebar di berbagai tempat, sebagai contoh pada sistem informasi perbankkan dan pemesanan tiket pesawat, yang mempunyai basis data yang tersebar di sejumlah cabangnya.

Basis data terdistribusi[1] menyediakan akses data secara transparan bagi pemakai (*user*), sehingga seolah-olah pemakai mengakses data lokal pada satu mesin, padahal data itu tersebar pada mesin-mesin yang lain. Proses pengaksesan data tersebut dilakukan secara otomatis oleh sistem, sehingga pemakai dibebaskan

[1] Ceri S and Pelagatti G, "Distributed Databases: Principles and Systems", New York: McGraw-Hill, (1984), p13.

untuk tahu dimana tempat data yang akan diakses, sehingga akan menambah kemudahan bagi pemakai untuk mendapatkan data yang diminta. Sistem basis data terdistribusi menggunakan data dan DBMS yang tersebar dalam berbagai macam sistem komputer, sehingga sistem ini memerlukan pula fasilitas komunikasi data dalam jaringan komputer untuk mendukung fasilitas *client/server*.

Prototipe basis data terdistribusi ini menggunakan bahasa Java, yang pada saat ini sedang populer karena filosofinya "*Write once, run anywhere*" (Tulis program sekali, jalankan di mana saja), merupakan dasar bagi pengembangan sistem ini. Dengan memanfaatkan *Java Data Base Connectivity (*JDBC) akan diperoleh kemudahan untuk melakukan perubahan bagian DBMS sesuai dengan yang diinginkan. Keunggulan-keunggulan tersebut dipakai dalam penelitian prototipe sistem manajemen basis data terdistribusi ini untuk mendukung terhadap sifat portabilitas yang tidak bergantung pada suatu jenis perangkat keras dan sistem operasi tertentu serta pengolahan data yang tersebar tidak tergantung pada suatu macam DBMS tertentu.

1.2. Perumusan Masalah

Prototipe sistem manajemen basis data terdistribusi merupakan kumpulan data dari satu sistem yang secara fisik tersebar pada beberapa lokasi dalam suatu jaringan komputer. Sistem basis data terdistribusi tersebut mengolah data yang berada pada mesin-mesin yang tersebar di tempat yang berlainan, sehingga pemakai sistem basis data tersebut seolah-olah mengakses sebuah sistem manajemen basis data tunggal.

Dengan menggunakan bahasa Java, maka sistem manajemen basis data terdistribusi yang dibuat tidak tergantung pada suatu jenis sistem operasi atau perangkat keras tertentu. Selain itu, dengan digunakannya *driver* JDBC akan mempermudah pemakai untuk menggantinya dengan suatu macam DBMS lainnya, sesuai dengan yang dibutuhkan tanpa harus melakukan kompilasi atau merombak ulang program sistem manajemen basis data terdistribusi dan aplikasi sistem informasi di sisi *client*.

1.3. Batasan Masalah

1. Prototipe basis data terdistribusi yang dibuat ditujukan untuk membangun di modul bagian komponen basis data terdistribusi (*distributed database component*/DDB). Komponen ini yang bertanggung jawab dari semua proses yang mengakses data pada lebih dari satu lokasi, dan mendapatkan informasi dari data yang tersebar dari katalog (*data dictionary*/DD). Katalog menyatakan informasi dari data yang tersebar dalam jaringan komputer. Bagian komponen basis data terdistribusi tersebut menggunakan bagian manajemen basis data (*database manajemen component*/DB) untuk menggolah basis data secara lokal, pada modul ini digunakan perangkat lunak DBMS yang telah ada untuk mengolah manajemen basis data terdistribusi tersebut. Sedang pada modul bagian komunikasi (*data communication component*/DC) menggunakan fungsi - fungsi jaringan komputer pada sistem operasi yang telah ada yang diakses menggunakan bahasa Java.

2. Sistem manajemen basis data terdistribusi ini juga bertanggung jawab atas sinkronisasi data antara data hasil replikasi data yang di komputer lain dan data utama. Data hasil replikasi yang dipunyai dari data utama yang berada pada komputer yang lain, harus di*update* secara terus-menerus, sehingga bisa digunakan dalam proses *query* data. Biasanya terjadinya penambahan data secara lokal menyebabkan perbedaan data replikasi dengan data utama, sehingga memerlukan proses sinkronisasi untuk mengatasi masalah tersebut.
3. *Data Dictionary* dan *Directory* diperlukan pada sistem manajemen basis data terdistribusi guna membuat sentralisasi secara logika dari data yang ada. *Dictionary* memberikan informasi tentang apa data itu dan bagaimana data itu diartikan (secara logika). Sedang *directory* menyediakan informasi dimana data tersebut secara fisik dapat ditemukan dan bagaimana ia dapat diakses.
4. Prototipe sistem basis data terdistribusi ini mendukung bahasa SQL dasar sesuai standar *grammar* SQL minimum[2] yaitu :
    1. *Data Definition Language* (DDL): CREATE TABLE dan DROP TABLE.
    2. *Data Manipulation Language* (DML) sederhana : SELECT, INSERT, UPDATE SEARCHED, dan DELETE SEARCHED.
    3. Ekspresi: sederhana (seperti A>B AND C OR D).
    4. Tipe data: CHAR, INTEGER, atau FLOAT.
5. Untuk meningkatkan kecepatan dilakukan proses optimasi, sehingga jumlah data yang harus ditransfer antar komputer dapat dihemat dan meningkatkan kecepatan komunikasi. Saluran komunikasi yang digunakan harus bebas dari terputusnya

[2] Microsoft, "ODBC 3.0 Reference", Microsoft Help File, (1996),Section "SQL Minimum Grammar".

hubungan saluran komunikasi tanpa persetujuan dari sisi penerima maupun pengirim, sehingga masalah sinkronisasi yang mungkin terjadi ialah masalah perbedaan data pada beberapa tuple data yang diakibatkan oleh akses data secara lokal.

1.4. Tujuan Penelitian

Adapun tujuan dilakukannya penelitian ini sebagai berikut :

1. Membuat prototipe sistem manajemen basis data terdistribusi yang dengan fleksibel dapat menggunakan perangkat lunak DBMS tersentralisasi (site tunggal) yang telah ada. DBMS yang telah ada tersebut dapat berbeda-beda *platform* atau perangkat keras, tetapi harus mempunyai *driver* JDBC sendiri-sendiri. Serta dapat digunakan di berbagai *platform* yang berbeda perangkat keras maupun sistem operasi.
2. Mengatur komunikasi dan masalah sinkronisasi pada tiap-tiap basis data yang ada.
3. Mengatasi masalah kecepatan yang terjadi ketika melakukan pemprosesan data yang tersebar di berbagai tempat yang lain.

1.5. Metodologi Penelitian

1. Studi literatur dari berbagai buku dan jurnal

   Pada studi literatur ini dicari dan dipelajari berbagai buku dan jurnal mengenai sistem manajemen basis data terdistribusi.

2. Perancangan algoritma program dan struktur data

Pada tahap ini ditentukan algoritma program yang akan digunakan lalu dirancang struktur datanya.

3. Pembuatan perangkat lunak

   Pada tahap ini dibuat perangkat lunak yang dimulai dengan dibuat modul-modul yang ada kemudian digabungkan secara keseluruhan.

4. Pengujian perangkat lunak

   Pada tiap-tiap modul yang telah dibuat diuji apakah sudah bekerja dengan benar dan kemudian modul-modul tersebut digabungkan untuk diuji keseluruhan perangkat lunak apakah dapat bekerja dengan baik.

5. Perbaikan untuk meningkatkan kinerja

   Pada tahap ini rutin-rutin perangkat lunak yang tidak perlu dapat dihilangkan dan jika perlu dilakukan perubahan atau penambahan rutin.

6. Pembuatan laporan

   Pada tahap ini ditulis laporan mengenai dasar teori, perancangan program serta analisa program.

1.6. Sistematika Penulisan

Penulisan Tugas Akhir ini dibagi menjadi tujuh bab, pembagiannya sebagai berikut :

BAB I PENDAHULUAN

Bab ini menjelaskan tentang latar belakang, permasalahan, tujuan, batasan permasalahan dan metodologi yang digunakan untuk menyelesaikan tugas akhir.

BAB II PENGENALAN BASIS DATA TERDISTRIBUSI

Bab ini menjelaskan mengenai basis data terdistribusi, konsep dasar, dan penggolongan yang ada pada basis data terdistribusi. Penggolongan ini dibedakan berdasarkan arsitektur dan non-arsitektur.

BAB III ALGORITMA-ALGORITMA BASIS DATA TERDISTRIBUSI

Bab ini menjelaskan metoda dan algoritma yang digunakan dalam prototipe sistem manajemen basis data terdistribusi ini.

BAB IV JAVA DATABASE CONNECTIVITY

Bab ini menjelaskan konsep-konsep yang ada pada *Java DataBase Connectivity* (JDBC).

BAB V PERANCANGAN DAN PEMBUATAN PERANGKAT LUNAK

Bab ini menjelaskan mengenai arsitektur, desain dan implementasi dalam bentuk struktur data dan prosedure dalam prototipe sistem manajemen basis data terdistribusi.

BAB VI UJI COBA DAN EVALUASI SISTEM

Bab ini menjelaskan cara-cara pelaksanaan uji coba, hasil uji coba dan evaluasi sistem yang membahas mengenai faktor-faktor yang mempengaruhi unjuk kerja sistem.

BAB VII KESIMPULAN DAN SARAN

Bab ini merupakan akhir dari penulisan buku ini berisi kesimpulan dan saran pengembangan berikutnya dari hasil tugas akhir ini.

BAB II

PENGENALAN BASIS DATA TERDISTRIBUSI

Dalam sistem basis data terpusat, semua komponen sistem berada pada sebuah komputer tunggal. Komponen tersebut berupa data, perangkat lunak DBMS, dan media penyimpanan seperti disk untuk penyimpanan basis data secara *on-line* dan *tape* untuk *backup*. Basis data terpusat dapat diakses dari jauh melalui terminal-terminal yang tersambung ke komputer tersebut. Dalam saat sekarang ini terjadi perubahan trend secara cepat menuju ke sistem komputer terdistribusi yang berupa beberapa *site* komputer yang dikoneksi melalui jaringan komunikasi. Perangkat lunak yang digunakan untuk implementasi sistem tersebut disebut sistem manajemen basis data terdistribusi (*Distributed Data Base Manajemen System* / DDBMS).

Dorongan untuk mendistribusikan data datang dari kebutuhan pemakai dan dorongan teknologi. Kebutuhan pemakai ditandai oleh bertumbuhnya informasi sistem dimana keamanan dan konsistensi yang memerlukan perawatan dan kemampuan otonomi lokal. Dorongan teknologi terutama pada bagian prossesor dan saluran komunikasi data komputer yang semakin cepat.

Selain itu dorongan untuk menggunakan DDBMS karena kebanyakan akses pemakaian data, hanya ke data lokal. Pengalaman menunjukan bahwa pada kenyataannya 80 % data diakses dari lingkungan lokal sedangkan 20% mengakses data di luar lingkungan lokal dari keseluruhan waktu yang ada[3]. Hal ini

[3] Atre S. "Distributed Database, Cooperative Processing, & Networking",McGraw-Hill,(1993), p.59.

mengakibatkan dengan memecah data dalam beberapa *site* lokal dan mengakses data di salah satu *site* lokal tersebut akan menyediakan kinerja yang lebih baik daripada menempatkan data hanya di satu lokasi pusat.

Pemakai perlu mempunyai aplikasi dan alat bantu cocok untuk basis data terdistribusi, untuk memecahkan berbagai masalah yang terjadi dalam pemakaian basis data terdistribusi, serta kemungkinan masalah yang terjadi antara pemakai. Pemecahan masalah-masalah tersebut meliputi optimasi untuk proses *query*, proses sinkronisasi dari berbagai cabang, alokasi data ke cabang-cabang, mengontrol akses data, fasilitas penyelamatan (*recovery*) dari berbagai kegagalan komponen sistem dan pemeliharaan data yang secara fisik letaknya tersebar.

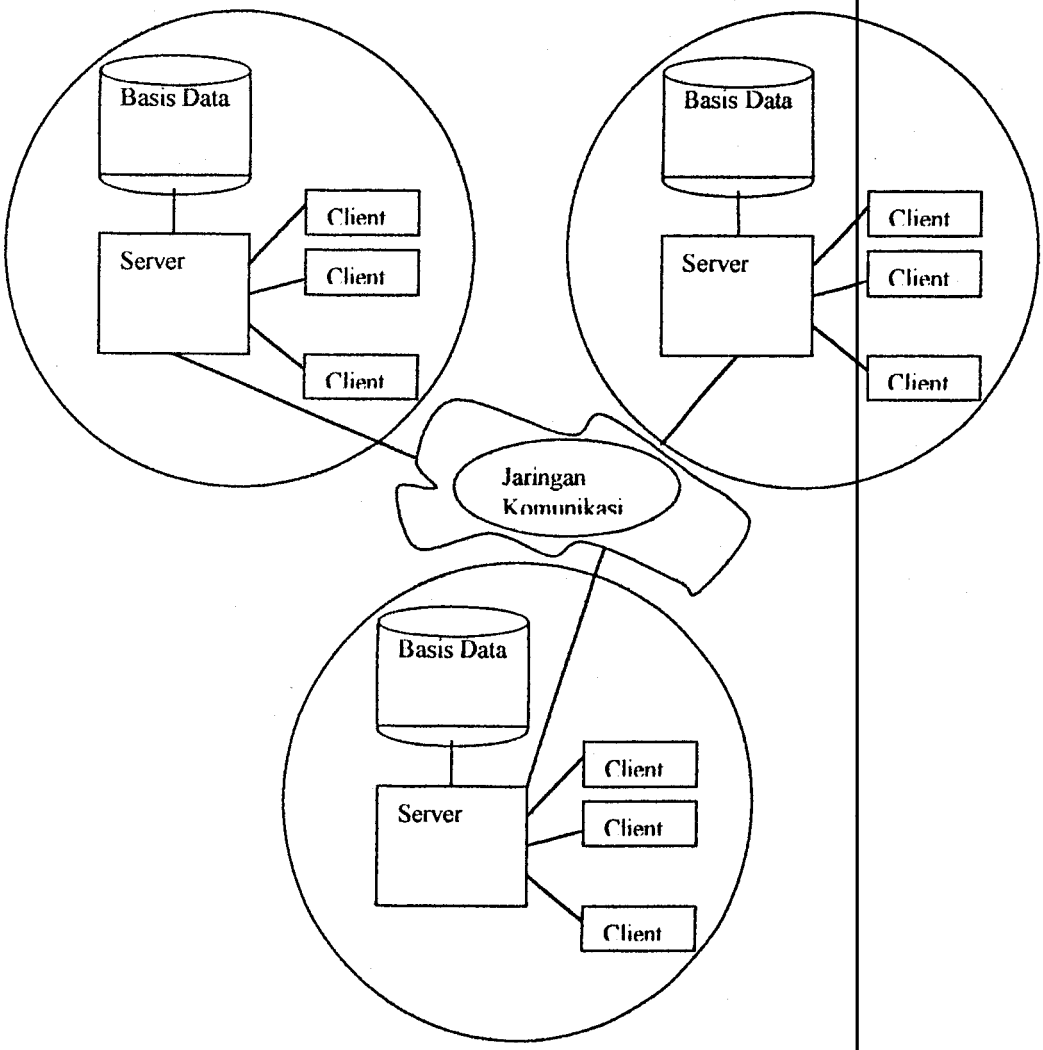
Pada dasarnya DDBMS menutup detail distribusi data yang terjadi pada keseluruhan proses yang ada. Fungsi itu memungkinkan pemakai untuk melakukan *query* secara global dan transaksi seperti yang dilakukan dalam basis data terpusat, tanpa perlu menentukan atau mengetahui *site-site* yang digunakan dalam proses *query* atau transaksi yang diberikan. Fungsi ini dinamakan transparansi distribusi (***distribution transparancy***).

2.1. Definisi Basis Data Terdistribusi

Basis data terdistribusi merupakan kumpulan dari data yang tersebar melalui beberapa komputer (*site*) yang berbeda dari suatu jaringan komputer[4]. Setiap *site* dari jaringan komputer mempunyai kemampuan memproses sendiri (*autonomous*) dan dapat menjalankan aplikasi lokal. Setiap *site* juga dapat berpartisipasi dalam eksekusi

[4] Ceri S, p.6.

dari paling sedikit satu aplikasi global, dimana dibutuhkan akses data dari beberapa *site* dengan menggunakan bagian sistem komunikasi. Contoh sistem basis data terdistribusi yang terpisah secara geografis dapat dilihat pada gambar 2.1.



Gambar 2.1 Basis data terdistribusi yang terpisah secara geografis dengan jaringan komputer

2.2. Keunggulan Basis Data Terdistribusi

Keunggulan yang merupakan ciri-ciri dari basis data terdistribusi[5] :

[5] Elmasri R. and Navathe S.B., "Fundamentals of Database Systems", Benyamin/Cummings, (1994), p.704.

- Ciri tersebar alami dari beberapa aplikasi basis data

Beberapa aplikasi basis data secara alami tersebar melalui lokasi berbeda. Sebagai contoh: sebuah perusahaan mempunyai lokasi di kota-kota yang berbeda, atau bank yang mempunyai beberapa cabang. Mereka menggunakan aplikasi basis data yang tersebar pada lokasi-lokasi tersebut. Beberapa **pemakai lokal** mengakses data hanya di lokasi tersebut, tetapi **pemakai global** seperti perusahaan pusat, memerlukan akses data ke berbagai lokasi.

- Menambah kepercayaan (*Reliability)* dan kesediaan (*Availability)*

Kedua potensi ini merupakan ciri-ciri basis data terdistribusi yang paling sering dibicarakan. Kepercayaan merupakan kemungkinan sebuah sistem tetap terjaga di waktu yang tertentu, sedang kesediaan merupakan kemungkinan suatu sistem yang secara terus-menerus tersedia selama interval waktu tertentu. Ketika salah salah satu *site* mungkin terjadi kegagalan sementara *site-site* lain dapat terus beroperasi, sistem masih dapat berjalan dan diakses datanya dengan baik. Untuk meningkatkan kedua ciri tersebut dapat dilengkapi dengan replikasi data dan perangkat lunak yang berada lebih dari satu *site*. Jika dibandingkan dalam sistem terpusat, kegagalan sebuah *site* tunggal membuat seluruh sistem tidak tersedia ke semua pemakai.

- Penggunaan data bersama-sama

Penggunaan data bersama-sama secara global dapat dilakukan bersamaan dengan akses dan kontrol data lokal. Data-data yang ada dapat diakses pemakai di *site* lain yang jauh melalui perangkat lunak DDBMS yang mengijinkan pengontrolan serta penggunaan data bersama secara terdistribusi.

- Memperbaiki kinerja

Basis data yang besar ukurannya didistribusikan ke berbagai *site-site* yang ada, basis

data tersebut dipecah ke dalam ukuran yang lebih kecil yang ditempatkan di *site-site* lokal. Sebagai hasilnya, pemakaian *query* lokal dan akses data transaksi di sebuah *site* lokal akan mengakibatkan kinerja yang lebih baik karena basis data lokal tersebut lebih kecil ukurannya. Sedang untuk perintah transaksi yang melibatkan lebih dari satu *site*, dengan tersebarnya data ke berbagai *site*, memungkinkan transaksi tersebut diproses secara paralel, sehingga mengurangi waktu proses.

2.3. Arsitektur Client-Server

Sistem *Client-Server* ini maksudnya ialah perangkat lunak aplikasi (*client*) mengirimkan perintah ke *server*, yang terpisah melalui jaringan komputer, yang akan memproses perintah tersebut lalu memberikan hasil proses ke *client*. Interaksi proses antara *client* dan *server* dalam memproses *query* SQL :

1. *Client* mengirimkan *query* ke *site server* yang sesuai.
2. *Server* memproses *query* dan mengirimkan hasil relasi ke *site client*.
3. *Client* memproses hasil *query* dan memberikan hasil tampilan yang sesuai pada aplikasi yang diakses oleh pemakai.

Dalam pendekatan ini *server* SQL dinamakan mesin ***back-end***, dan *client* disebut mesin ***front-end***.

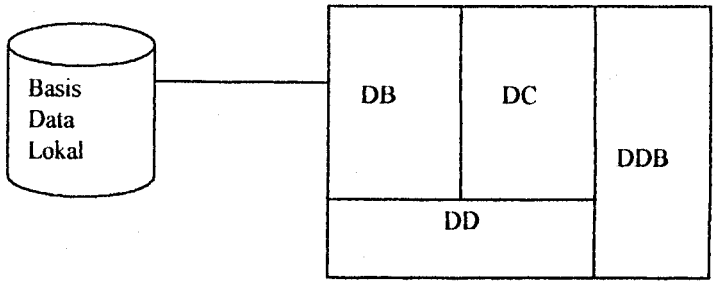
Khusus pada DDBMS, ia dibagi menjadi tiga level modul perangkat lunak :

1. Perangkat lunak *server* bertanggung jawab untuk manajemen data di suatu *site*, yang mirip seperti perangkat lunak DBMS tersentralisasi.
2. Perangkat lunak *client* yang digunakan untuk mengakses data dari DDBMS dan menerima hasil permintaan, serta menampilkannya kepada pemakai.
3. Perangkat lunak komunikasi (sering kali mengunakan sistem operasi terdistribusi)

menyediakan sarana komunikasi yang digunakan oleh *client* untuk mengirimkan perintah dan data ke berbagai *site* yang diperlukan.

Secara umum komponen-komponen yang ada dalam sistem basis data terdistribusi (gambar 2.2.) adalah :

1. Komponen manajemen basis data (DB)
2. Komponen komunikasi data (DC)
3. Komponen katalog data (DD), yang menyatakan informasi tentang pendistribusian data dalam jaringan.
4. Komponen basis data terdistribusi (DDB)



Gambar 2.2. Komponen dari DDBMS

2.4. Klasifikasi Sistem Basis Data Terdistribusi

DBMS terpusat merupakan sebuah sistem yang melakukan manajemen sebuah basis data (database/DB) tunggal, sedangkan sebuah DDBMS merupakan DBMS yang melakukan manajemen beberapa basis data. Untuk membedakan jenis-jenis sistem basis data terdistribusi diperlukan klasifikasi[6] yang jelas antar DDBMS yang ada. Kata global dan lokal sering digunakan ketika membicarakan DDBMS (atau pada sistem terdistribusi pada umumnya) untuk membedakan antara aspek yang

[6] Bell D.A., p.44.

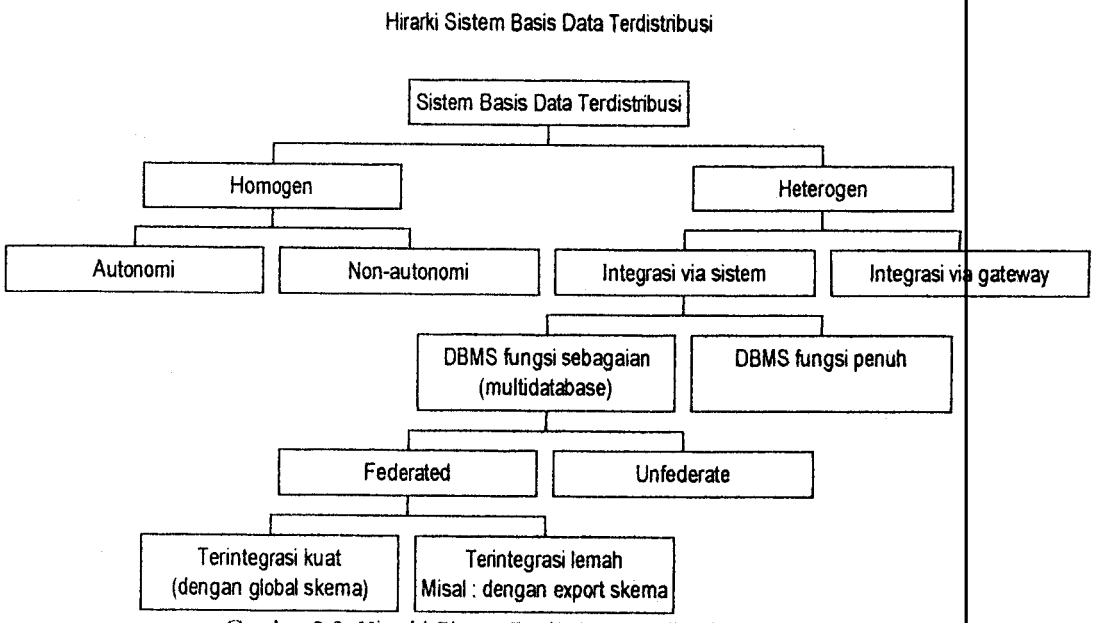
mengacu ke sebuah *site* tunggal (lokal) dan yang mengacu ke sistem secara keseluruhan (global). Sebagai contoh, basis data lokal mengacu ke basis data yang ditempatkan di salah satu *site* dalam jaringan, sedang basis data global mengacu ke integrasi logikal dari semua basis data lokal.

2.4.1. Klasifikasi DDBMS Secara Arsitektur

Sistem basis data terdistribusi dibagi dalam dua tipe, dari filosofi berbeda, dan didesain untuk memenuhi kebutuhan berbeda:

1. Sistem manajemen basis data terdistribusi homogen
2. Sistem manajemen basis data terdistribusi heterogen

Hirarki penuh dari sistem data terdistribusi ditunjukan dalam gambar 2.3.



Gambar 2.3. Hirarki Sistem Basis Data Terdistribusi berdasarkan arsitektur

Sebuah DDBMS **homogen** mempunyai beberapa koleksi data dan ia mengintegrasi beberapa sumber data. Ia dibagi menjadi dua kelas tergantung dari sifat otonominya. Istilah **otonomi** ini untuk memberikan kepada sistem hak untuk melakukan

kontrol secara lokal. DDBMS homogen menyerupai sebuah DB terpusat, tetapi disamping menyimpan semua data dalam satu *site*, data didistribusikan melalui sejumlah *site* dalam jaringan komputer. Skema global merupakan gabungan dari semua deskripsi data lokal (tidak dinyatakan hanya dalam sebuah skema).

Sistem **heterogen** merupakan kelas dengan karakteristik penggunaan DBMS yang berbeda di cabang-cabang lokal. Dipecah menjadi dua subkelas utama: **terintegrasi penuh dengan sistem**, dan yang menyediakan tambahan eksternal yang dinamakan ***gateway***, untuk mengijinkan sistem yang lainnya digabung. Sub-kelas tersebut dipecah dalam sub-kelas yang lain dengan menyediakan subset khusus dari fungsi yang diharapkan dari beberapa DBMS dan yang menekankan lebih aspek pragmatis ke pengatur data kolektif, seperti konversi diantara sistem dan beberapa ciri-ciri khusus untuk meningkatkan kinerja.

Sistem Manajemen Multi Basis Data (***Multi Database Manajemen Sistem*/MDBMS**) mempunyai beberapa DBMS sekaligus yang menggunakan basis data yang telah ada dan memungkinkan untuk berbeda tipe struktur data. Proses integrasi itu dilakukan oleh beberapa perangkat lunak sub-sistem. Keseluruhan arsitektur dari sebuah MDBMS, membedakan antara pemakai lokal dan global. MDBMS mengintegrasi sumber daya yang telah ada dan yang heterogen. Ciri-ciri khusus dari sistem ini dimana pemakai lokal dapat tetap melanjutkan untuk mengakses basis data lokal dalam sistem normal lokal dalam cara biasanya, sehingga tidak terganggu oleh kehadiran MDBMS.

Selain itu terdapat ***federated*** dan ***unfederated*** MDBMS. Dalam *unfederated* sistem, tidak ada pemakai lokal dan subkelas ini agak kabur batasannya. Sistem *federated* dipecah ke sebuah skema global terikat erat (***tight Coupled***) dan yang tidak terlalu terikat

erat (***loosely coupled***).

Skema global merupakan sebuah pandangan secara logikal dari semua data yang ada dalam MDBMS. Ia hanya merupakan penggabungan dari semua skema lokal, sementara DBMS lokal bebas untuk menentukan bagian mana dari basis data lokal yang mereka harapkan untuk berkontribusi dari skema global. Kebebasan ini dinamakan sebagai **otonomi lokal**. Skema *auxiliary* dalam MDBMS menyatakan aturan yang mengatur pemetaan diantara level lokal dan global. Sebagai contoh, aturan untuk konversi unit mungkin diperlukan ketika satu *site* menyatakan jauh dalam kilometer dan yang lain dalam mil. Aturan untuk mengatasi nilai 'null' mungkin penting dimana satu *site* menaruh informasi tambahan yang tidak ditempatkan *site* yang lain. Contoh lainnya satu *site* menaruh nama, alamat rumah dan nomor telepon dari pekerjanya, sedang yang lain hanya menaruh nama dan alamat.

Loosely coupled MDBMS mempunyai karakteristek penting yaitu mereka tidak mempunyai skema konseptual global. Membuat sebuah skema konseptual global merupakan suatu hal yang sulit dan tugas yang komplek yang melibatkan pemecahan masalah semantik dan sintaktik yang berbedaan antara *site-site* yang ada. Sering kali perbedaan ini terlalu banyak sehingga *site* tersebut tidak dapat dilibatkan dalam skema global.

Terdapat dua pendekatan utama untuk membuat *loosely coupled* MDBMS. Salah satunya bagi pemakai untuk membuat pandangan sendiri melalui skema konseptual lokal, menggunakan bahasa *query* seperti SQL. Sedangkan yang lain, basis data lokal dapat mendefinisikan kontribusi mereka untuk basis data *federated* oleh arti dari skema export (analogi dari skema partisipasi dari MDBMS).

2.4.2. Klasifikasi DDBMS Non-Arsitektural

Klasifikasi DDBMS non-arsitektural ini mengkategorikan menurut tingkat heterogen sebuah sistem, metode dari distribusi data dan perluasan dari otonomi lokal. Sifat **heterogen** dalam DDBMS dapat menimbulkan beberapa level perbedaan dalam sistem, termasuk didalamnya perbedaan perangkat keras di *site-site* yang ada yang berbeda sistem operasi dan protokol jaringan. Disisi basis data, sifat heterogen dapat timbul ketika DBMS lokal yang berbeda digunakan dengan data model yang sama (seperti INGRES atau Oracle) atau berdasar di model data berbeda (relational, hirarki atau *network*). Juga jika meskipun DBMS lokal sama dapat terjadi perbedaan arti sematik.

Metoda pendistribusikan data antar *site* dapat dibedakan dengan cara yang berbeda-beda. Data mungkin terreplikasi penuh (*fully replicated*), dimana semua basis data global disimpan dalam setiap cabang. Replikasi penuh digunakan ketika toleransi untuk kesalahan dan kinerja saat proses *query* sangat penting, data juga harus selalu tersedia setiap waktu serta disana tidak ada tundaan komunikasi (untuk pemanggilan).

Dalam kasus tertentu, basis data global dapat benar-benar terpecah (*fully partitioned*), juga tidak ada replikasi. Seperti situasi yang sering terjadi dengan MDBMS dimana basis data lokal yang telah ada dan diintegrasi dalam model dari bawah ke atas. Replikasi sebagian (*partial replication*) umumnya dibutuhkan ketika bagian dari basis data global sering diakses dari sejumlah *site* berbeda dalam jaringan.

Tingkatan dari **otonomi lokal** didukung dalam sebuah lingkungan DDB yang merupakan faktor lain yang penting. Dimana sebuah sistem mengijinkan otonomi penuh pada cabang-cabang, proses integrasi memang akan menjadi lebih susah. Otonomi sering membandingkan antara kontrol terdistribusi dengan data terdistribusi. Jika yang satu tidak

ada lokal otonomi, hal ini merupakan kontrol global penuh. Seperti sebuah situasi dapat ditemukan dalam lingkungan basis data terdistribusi homogen yang didesain secara dari atas ke bawah. Hal ini merupakan sebuah sistem yang tidak mempunyai pemakai lokal sedangkan basis data lokal diatur oleh DBMS lokal, sedangkan DBMS lokal tidak beroperasi sendiri.

Sebaliknya dalam MDBMS mempunyai pemakai lokal dan global dan otonomi lokal menjamin bahwa pemakai lokal mengakses basis data lokal sendiri berdiri sendiri, dan tidak saling mempengaruhi, eksistensi dari MDBMS data dan pemakai global. Penggolongan sifat-sifat otonomi :

Otonomi desain merupakan cara yang paling dasar ketika ia memberikan *site* lokal kebebasan desain dan implementasi termasuk memutuskan isi informasi dari DB, seleksi dari data model dan pilihan dari struktur penyimpanan. Dalam MDB dimana tujuan untuk mengintegrasi basis data yang telah ada, otonomi desain lokal sangat dibutuhkan.

Otonomi partisipasi memberikan *site* lokal untuk memilih bagaimana partisipasi dalam sistem terdistribusi, kapan dan apa data yang mengkontribusi. *Site* lokal secara ideal harus membebaskan akses data yang datang dan pergi sesuai keperlukan.

Otonomi komunikasi memberikan *site* kebebasaan untuk menentukan bagaimana dan dibawah arti apa dalam berkomunikasi dengan *site* lain di dalam jaringan. Dalam praktek, dari fungsi ini paling banyak diambil oleh jaringan komputer sendiri.

Otonomi eksekusi memberikan kesanggupan *site* lokal untuk menentukan bagaimana dan apa untuk memproses operasi lokal, untuk menaruh, memanggil dan update data lokal, misalnya manajer transaksi lokal harus mempunyai kebebasan untuk membatalkan transaksi global jika ia konflik dengan transaksi lokal.

BAB III

ALGORITMA-ALGORITMA BASIS DATA TERDISTRIBUSI

Pada bab ini dibahas mengenai algoritma-algoritma apa saja yang ada dalam sistem manajemen basis data terdistribusi. Algoritma-algoritma yang dibahas meliputi fragmentasi data, replikasi dan alokasi data, optimasi, *concurrency control*, masalah query, *recovery* dan keamanan yang digunakan dalam penelitian Tugas Akhir ini.

3.1. Fragmentasi Data

Desain dari pecahan (*fragment*) adalah problem utama yang harus dipecahkan dalam model desain dari atas ke bawah dari pendistribusian data. Tujuan dari desain fragmentasi untuk menentukan fragmentasi basis data agar tidak saling bertumpukan, fragmentasi disini merupakan "unit-unit logika dari alokasi". **Skema fragmentasi** menyatakan bagaimana relasi global dipecah melalui basis data-basis data lokal. **Skema alokasi** menyatakan di *site* mana pecahan (*fragment*) ditempatkan.

Desain *fragment* akan mengelompokkan data menurut tupel (dalam fragmentasi horisontal) atau atribut (dalam fragmentasi vertikal) atau menurut keduanya (dalam fragmentasi campuran) yang mempunyai ciri yang sama untuk alokasinya. Setiap kelompok dari tupel atau atribut yang mempunyai ciri sama akan menjadi sebuah *fragment*. Beberapa metoda digunakan untuk alokasi data agar mengelompokkan mereka secara bersama-sama.

Fragmentasi horisontal dari sebuah relasi merupakan subset dari tupel-tupel

dalam relasi tersebut. Tupel-tupel dari fragmentasi horisontal dinyatakan dengan kondisi nilai dari satu atau lebih atribut dari relasi, sering kali hanya sebuah atribut tunggal yang digunakan. Setiap fragmentasi horisontal dalam relasi R dapat didefinisikan dengan operasi $\sigma_{Ci}(R)$ dalam matematika relasional. Kumpulan *fragment* horisontal yang memenuhi kondisi C1, C2, ..., Cn termasuk di semua tupel dalam R, maka setiap tupel dalam R yang memenuhi (C1 or C2 or ...or Cn) dinamakan **fragmentasi horisontal lengkap** dari R. Fragmentasi horisontal juga mempunyai sifat **disjoin** dimana tidak ada tupel yang memenuhi (Ci and Cj) untuk $i \neq j$. Untuk merekonstruksi relasi R dari fragmentasi horisontal lengkap, kita hanya memerlukan operasi UNION ke *fragment-fragment* yang ada.

3.2. Replikasi Dan Alokasi Data

Replikasi berguna untuk memperbaiki sifat kesediaan (*availability*) dari data. Replikasi yang terdapat pada keseluruhan basis data di setiap *site* dalam sistem terdistribusi disebut basis data terdistribusi **terreplikasi penuh** (***fully replicated***). Penambahan sifat kesediaan ini terjadi karena sistem tetap dapat melanjutkan operasi selama paling sedikit satu *site* tetap jalan. Selain itu replikasi berguna untuk memperbaiki kinerja dari pemanggilan query global, ini dikarenakan query dapat dilakukan secara lokal dari *site* tunggal. Ketidakuntungan dari sistem terreplikasi penuh ia dapat memperlambat sistem saat operasi query tipe *update* dilakukan, karena perlu dilakukan *update* di semua *site* yang terdapat data replikasi agar data-data tersebut tetap sama. Replikasi penuh memerlukan kontrol *concurrency* dan teknik *recovery* yang sulit.

Selain replikasi penuh ada metode **tanpa replikasi** di mana setiap *fragment*

ditaruh hanya di satu *site*. Dalam kasus ini semua *frament* harus ***disjoin***, kecuali untuk perulangan primary key pada fragmentasi vertikal (atau campuran). Ini disebut alokasi yang tidak saling tertumpuk (***nonredundant allocation***).

Selain kedua cara diatas terdapat cara lain yang disebut replikasi sebagian (***partial replication***) yaitu beberapa *fragment* dari basis data mungkin terreplikasi sedangkan beberapa yang lain tidak. Jumlah kopi dari setiap *fragment* dapat berkisar dari satu ke jumlah total dari *site* dalam sistem terdistribusi. Deskripsi dari replikasi *fragment* seringkali disebut **skema replikasi**.

Pemilihan *site* dan tingkat replikasi tergantung tujuan kinerja dan *availability* dari sistem dan frekuensi dari transaksi yang dikirimkan ke setiap *site*. Sebagai contoh, jika diperlukan *avilability* tinggi dan transaksi harus sering dikirimkan ke beberapa atau semua *site*, sebuah basis data terreplikasi penuh merupakan pilihan yang baik. Sedang jika transaksi sering mengakses dari beberapa *fragment* basis data tertentu, data yang sering diakses di beberapa *site*, dapat di replikasi di *site* tersebut. Jika perlu update dilakukan, ini sangat berguna untuk membatasi replikasi. Untuk menemukan cara optimal atau solusi yang baik untuk alokasi data terdistribusi merupakan problem optimasi yang komplek.

Ada dua macam algoritma dalam replikasi yaitu *store-dan-forward* (***asynchronous***) dan real-time (***synchronous***). Berikut ini beberapa ciri-ciri replikasi synchronous dan asynchronous :

Synchronous

- Perubahan di salah satu *site* dengan cepat berakibat di *site-site* yang lain, sehingga selalu mempunyai informasi yang *up-to-date* di semua *site*.

- Data yang sama dapat diupdate dalam banyak *site*, tanpa perlu kuatir terjadinya konflik saat proses update data.
- Jika terjadi kegagalan jaringan komputer di *site* yang terreplikasi synchronous, akan terjadi kegagalan untuk melakukan update lokal sampai kerusakan jaringan diperbaiki atau memutuskan hubungan dengan *site* yang gagal (*down*) dari lingkungan replikasi.
- Update akan mempunyai waktu respon yang lambat, sebab harus menunggu respon dari semua *site*, sebelum commit atau roll back sebuah transaksi.

Asynchronous

- Kegagalan pada salah satu *site* atau lebih dalam jaringan komputer tidak akan menahan *site* yang lain dari perintah query atau replikasi data secara lokal.
- Respon waktu perintah query tipe update lebih cepat, sebab tidak perlu menunggu respon dari *site* yang lain.
- Waktu interval dapat disesuai menurut kebutuhan, jika ingin "mendekati *real-time*" replikasi dilakukan dengan menggunakan waktu interval yang singkat.
- Perubahan di salah satu *site* tidak secara langsung berakibat pada *site* yang lain, sehingga hasil query dapat tidak konsisten.
- Konflik ketidaksamaan data dapat terjadi di beberapa *site*. Konflik ini tidak akan terdeteksi sampai operasi update perubahan dilakukan.

3.3. Optimasi

Tujuan pelaksanaan optimasi terutama pada proses query untuk meminimalkan jumlah penggunaan sumber daya. Sumber daya yang paling kritis dalam sistem manajemen basis data adalah :

- CPU
- *Bus I/O*
- Saluran telekomunikasi

Bus I/O berperanan pada saat pemindahan dari tempat penyimpanan *secondary* ke penyimpanan utama dan kemudian digabungkan dengan data yang menempati penyimpanan sementara (*temporary*). Sedangkan pada saluran telekomunikasi karena kecepatannya yang lambat, sering kali dapat membuat sumber daya-sumber daya yang ada menjadi terhambat (terjadi *bottleneck*) sehingga ia sering menjadi media yang paling lambat untuk proses-proses yang ada. Peran CPU juga berpengaruh, tetapi dengan semakin bertambahnya kecepatan prossesor-prossesor yang ada sekarang ini sering diassumsikan prossesor lebih berkecepatan tinggi dibandingkan dengan saluran telekomunikasi dan bus I/O.

Optimasi juga bertujuan untuk meminimalisasi waktu respon saat query dilakukan dengan memberikan metode paralelisme yaitu dengan menggunakan banyak prossesor dalam jaringan komputer, dengan demikian satu proses dapat diproses secara bersama-sama sekaligus. Dalam sistem terdistribusi transmisi pesan dalam saluran komunikasi data diperlukan untuk mengontrol evaluasi query dalam sesuai urutan yang pengeksekusian. Transmisi data juga diperlukan untuk melewatkan hasil dan sebagian hasil diantara *site-site*.

Diantara operasi relasional dasar (*select, join, project*) yang paling memerlukan banyak biaya dalam arti waktu dan usaha ialah operasi *join*. Metode utama dalam melakukan optimasi operasi *join* dalam basis data terdistribusi dilakukan dengan menggunakan operasi *semi-join*. Operasi *semi-join* mengakibatkan penambahan pemprosesan data secara lokal dibandingkan dengan operasi *join* biasa, tetapi lebih menghemat biaya perpindahan data antar *site*.

Ia menggunakan operasi *project* dari tupel-tupel dari suatu relasi lalu melakukan *join* di salah satu relasi lainnya, misalnya R_1, dari pasangan (R_1, R_2) dari relasi yang akan dilakukan operasi *join*.

Notasi yang digunakan adalah $\lhd$ untuk operasi *semi-join* sebagai kondisi tambahan dari operasi *join*. Jika Π_i merupakan operasi *project* dari atribut R_i dan untuk operasi *natural join* dengan notasi $\rhd\lhd$ sebagai tambahan operasi *join* sebagai $\lhd$, maka *semi-join* dapat didefinisikan berikut :

$R_2 \lhd R_1 = \Pi_1 (R_1 \rhd\lhd R_2)$

Rumus ini dapat dengan mudah diganti oleh bentuk yang sama :

$R_2 \lhd R_1 = R_1 \rhd\lhd (\Pi R_2)$

Dilakukan operasi Π hanya kepada atribut yang berpengaruh pada operasi *join*.

Representasi kedua mempunyai keuntungan lebih besar dibandingkan dengan yang pertama sebagai contoh, jika R_1 dan R_2 berada di *site* berbeda A dan B dan eksekusi operasi *join* untuk R_1 dan R_2 dilakukan pada *site* ketiga C. Saat eksekusi (Π R_2), tranmisi dan operasi *join* lokal di A (memberikan hasil dari *semi-join*) dinamakan **fase reduksi** dari sebuah operasi *join* terdistribusi. Lalu diikuti oleh **fase proses** sederhana, dimana kedua relasi terreduksi di A dan B dikirimkan ke *site* eksekusi seperti C dan akhirnya

dilakukan operasi *join*. Metode ini dapat dilakukan untuk mengurangi biaya komunikasi ketika ketika sebuah operasi *join* yang lengkap dieksekusi di C. Pertama-tama R_1 dan R_2 keduanya merupakan relasi 'terreduksi *semi-join*' seperti diatas, dan kedua sub-result tersebut dikirim ke C untuk dieksekusi dengan operasi *join*.

Jika R_1 merupakan tabel untuk operasi *semi-join* dan hasil dari *semi-join* diberikan ke *site* ketiga C. prosedur ini mengikuti :

Procedure semi-join[7]***:***

Begin

Langkah 1. Lakukan operasi project dari atribut yang berpengaruh pada operasi join dari R_2 di B (ΠR_2), setelah melakukan operasi select sesuai yang diperlukan

Langkah 2. Kirimkan (ΠR_2) ke A

Langkah 3. Lakukan operasi semi-join dari R1 di A

Langkah 4. Pindahkan hasil ke C

End.

Keuntungan dari algoritma *semi-join* terbesar ketika (Π R_2) cukup kecil (maka langkah 3 mengurangi jumlah tupel-tupel dari kedua *site-site* lokal dan ditransmisikan ke *site* hasil) dan harga pemprosesan lokal lebih kecil daripada harga transmisi. Sifat selektivitas dari operasi *semi-join* membuatnya cocok untuk digunakan, dimana jika operasi *join* dapat menambah jumlah dari basis data, sedangkan operasi *semi-join* tidak pernah demikian.

Ketika operasi *join* selesai dilakukan di A atau B, buat R_i^1 sebagai hasil dari operasi *project* dari R_i ke kolom *join*, dan buat $|R_i \triangleleft R_j|$ menjadi ukuran dari hasil *semi-join*. Sebuah operasi *semi-join* akan mempunyai harga transmisi lebih rendah daripada sebuah *join* konvensional jika harga inisialisasi pesan (diperlukan biaya kehadiran pesan

---

[7] *Bell D.A.,p.138.*

tambahan untuk operasi *semi-join*) yang ditambahkan dimana *min* ($|R_2^1|+|R_2 \lhd R_1|$, $|R_1^1|+|R_1 \lhd R_2|$) adalah lebih kecil dari pada *min* ($|R_1|, |R_2|$). Perluasan dari metoda operasi *join* dengan *semi-join* ini dengan menggunakannya lebih banyak dari dua relasi, disini fase reduksi dari evaluasi query lebih objektif digunakan dengan merubah sebuah query ke urutan perintah yang efisien dari operasi *semi-join*.

Operasi *semi-join* dikatakan memberikan keuntungan jika ia mereduksi ukuran basis data yang harus ditransmisikan. Kesusahan utama dari metode ini adalah menemukan sebuah harga terbaik untuk waktu respon dari *semi-join* dari query yang diberikan. Keuntungan operasi semi-join terlihat jika ia merupakan 'berbiaya-efektif' (karena, keuntungan lebih besar daripada biayanya). Suatu metoda heuristik dapat digunakan untuk eksekusi query tersebut .

3.4. Concurrency control

Concurrency Control dalam sistem manajemen basis data, merupakan bagian yang menjamin eksekusi dari beberapa transaksi yang berjalan bersama-sama, dengan tetap menjaga sinkronisasi item-item data yang diakses. Pada bagian ini dibahas mengenai transaksi, metoda manajemen transaksi dan dua fase *commit*.

3.4.1. Transaksi

Modul yang bertanggung jawab untuk *concurrency control* dalam urutan langkah-langkah query pada sebuah DBMS dinamakan ***scheduler***. Komunikasi antara program aplikasi dan *scheduler* dilakukan melalui **manager transaksi**. **Transaksi** disini

merupakan urutan dari aksi-aksi, yang dikeluarkan oleh pemakai tunggal / program aplikasi, yang harus dilakukan sebagai sesuatu yang tidak terpisahkan. Dalam *concurrency control* sering kali digunakan teori ***serializability*** yang berarti transaksi yang berjalan bersama-sama memberikan hasil yang sama, jika transaksi-transaksi tersebut dijalankan berurutan (***serial***).

Efek dari operasi ***rollback*** adalah untuk mengembalikan basis data ke status yang sama seperti sebelum transaksi dimulai dan menjadikan status sistem sekarang menjadi konsisten. Disini ada 4 ciri-ciri dari transaksi yang dinamakan **A.C.I.D**[8] :

- ***Atomicity*** : 'semua atau tidak sama sekali', sebuah transaksi merupakan sesuatu yang tidak terbagikan.
- ***Consistency*** : transaksi melakukan transformasi basis data dari satu status konsisten ke status konsisten yang lain.
- ***Independence*** : transaksi mengeksekusi secara berdiri sendiri tanpa saling terpengaruh satu sama yang lain.
- ***Durability*** (juga dinamakan ***persistence***) : efek dari transaksi yang lengkap (yang telah commit) secara permanen disimpan dalam basis data dan tidak dapat dibatalkan.

Ada tiga metode prinsip dalam *control concurrency* yaitu *locking*, *timestamp* dan *optimistic*. Metoda ***locking*** menggunakan penguncian (*lock*) item sebelum item tersebut dipakai dalam suatu transaksi, agar item tidak tertimpa atau digunakan oleh transaksi lain, sebelum transaksi yang melakukan penguncian selesai. Jika digunakan metoda *locking*

[8] Bell D.A.,p.165.

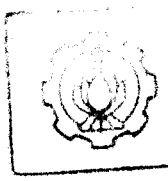
akan ada kemungkinan terjadinya ***deadlock*** yang berarti terjadi peristiwa saling menunggu, misalnya jika ada transaksi A yang berusaha melakukan operasi *lock* pada item X, sedangkan item tersebut telah di-*lock* oleh transaksi B yang akan melakukan operasi *lock* pada item Y, padahal item tersebut telah di-*lock* oleh transaksi A. Sedang metoda ***optimistik*** merupakan metoda yang jarang digunakan dalam basis data terdistribusi pada umumnya.

Metode *timestamp* dari concurency control agak berbeda dari metode *locking*. Disini tidak ada *lock* (pengunci) yang dilibatkan dan karena itu juga tidak akan terjadi *deadlock*. Metode *locking* biasanya membuat konflik antar transaksi yang ada untuk saling menunggu. Dengan metode *timestamp*, tidak ada yang menunggu, karena transaksi yang terlibat konflik dengan mudah di *rollback* dan di *restart* ulang.

Pada saat komunikasi data, sering terjadi bebearapa kegagalan. Kegagalan-kegagalan yang ada itu harus dideteksi dan dilaporkan dalam DDBMS, agar tidak mengganggu proses transaksi. Proses identifikasi masalah agak susah karena untuk menentukan apakah *site* tersebut *down* perlu harus mengirim beberapa *message* tambahan dengan *site-site* yang lain. Sebagai contoh, *site* X mengirim *message* ke *site* Y dan berharap sebuah respon dari Y tetapi ia tidak menerimanya. Terdapat beberapa kemungkinan-kemungkinan yaitu:

1. Message tidak terkirimkan ke *site* Y, sebab terjadi kesalahan komunikasi.
2. *Site* Y *down* dan tidak dapat meresponi.
3. *Site* Y jalan dan mengirimkan respon, tetapi respon tidak dikirimkan.

Tanpa mengirim informasi atau message tambahan, akan susah untuk



menentukan apa yang sesungguhnya terjadi. Dua fase *commit* digunakan untuk mencegah terjadinya kesalahan saat penyimpanan data, dimana jika dideteksi ada salah satu atau lebih *site* yang gagal saat proses *update* transaksi, maka proses *update* tidak akan dilakukan.

3.4.2. Metode Timestamp

Tujuan dasar dari metode *timestamp* adalah untuk mengurutkan transaksi global menurut usia/urutan transaksi. Transaksi yang lebih dulu masuk yaitu dengan nilai *timestamp* yang kecil, mendapat prioritas ketika terjadi konflik. Jika sebuah transaksi berusaha melakukan operasi baca atau tulis sebuah item data, tetapi operasi baca atau tulis itu hanya diijinkan untuk diproses, jika operasi *update* terakhir dari data item telah dilakukan oleh transaksi yang lebih dulu, kalau tidak permintaan transaksi di*restart* dan diberikan sebuah nilai *timestamp* baru. Nilai *timestamp* baru harus diberikan kepada transaksi yang di*restart*, untuk mencegahnya secara terus-menerus membuat permintaan *commit* yang ditolak. Metode *timestamp* memproduksi urutan langkah-langkah yang mempunyai sifat serial (*serializable*), yang sama dengan *schedule* serial yang didefinsisikan oleh nilai *timestamp* dari operasi *commit* transaksi yang berhasil.

Nilai *timestamp* digunakan untuk mengurutkan antara sebuah transaksi dengan transaksi yang lain. Setiap transaksi diberikan sebuah nilai *timestamp* yang unik ketika ia dijalankan, maka tidak akan ada dua transaksi mempunyai nilai *timestamp* yang sama. Dalam DBMS tersentralisasi, dapat dibuat sederhana dengan menggunakan jam pada sistem. Nilai *timestamp* dari transaksi dengan mudah menggunakan nilai dari jam pada sistem ketika transaksi dimulai. Sebagai alternatif yang lain, sebuah *counter* global

sederhana, atau urutan generator angka, dapat digunakan, yang beroperasi seperti mekanisme pengambil antrian tiket. Ketika sebuah transaksi dijalankan, ia diberi nilai selanjutnya dari *counter* transaksi, lalu counter dinaikkan. Untuk menghindari pembuatan dari nilai *timestamp* yang sangat besar nilai counter dapat secara periodik di reset kembali ke nol.

Sifat *atomicity* dari transaksi-transaksi dengan metoda *timestamp*, dapat dilihat saat melakukan operasi *commit* oleh transaksi untuk membuat *update* permanen dan dapat terlihat oleh transaksi yang lain (untuk menjamin sifat transaksi *atomicity* dan *durability*).

Transaksi yang telah dapat melakukan operasi *commit*, tidak pernah dapat dibatalkan kembali. Dengan protokol berbasis *locking*, *atomicity* transaksi dijamin oleh *write-locking* semua item-item sampai waktu perintah *commit* dan dengan dua fase *locking* secara khusus, semua *lock* dilepas bersamaan. Metoda *timestamp* tidak mempunyai kemungkinan untuk mencegah transaksi lain melihat operasi update yang baru dilakukan sebagian oleh transaksi ini, karena tidak ada penguncian (*lock*). Pendekatan lain dengan menutupi operasi *update* yang baru dilakukan sebagian dari transaksi-transaksi yang ada, dilakukan dengan menggunakan metoda *pre-writes* (*deferred update*). Operasi *update* dari transaksi yang belum commit tidak dapat ditulis langsung ke basis data. Tetapi ditulis ke buffer-buffer, dan hanya di lakukan operasi penulisan ke basis data ketika transaksi commit. Pendekatan ini memberikan keuntungan ketika sebuah transaksi dibatalkan atau di*restart*, tidak perlu ada perubahan fisik yang dibutuhkan pada basis data.

Basic Timestamping

Untuk mengimplementasikan metoda *basic timestamp* untuk DBMS berikut ini definisi variabel-variabel yang dibutuhkan :

Untuk setiap data item x

ts (read x) = timestamp dari transaksi yang terakhir membaca item data x

dan

ts (write x) = timestamp dari transaksi yang terakhir me-update item data x

Untuk setiap transaksi Ti

ts(Ti) = timestamp diberikan ke transaksi Ti ketika ia dijalankan

Metoda *timestamp* ini menggunakan sistem *pre-writes* untuk menjamin sifat *atomicity* transaksi, transaksi sesungguhnya melakukan operasi *pre-write* ke buffer daripada menulis ke basis data. Penulisan fisik ke basis data hanya dilakukan saat perintah *commit* dilakukan. Jika sekali diputuskan untuk melakukan operasi *commit* sebuah transaksi, sistem harus menjamin penulisan data dan ia tidak dapat dibatalkan.

Algoritma operasi *pre-write* dengan basic timestamp.

begin

Ti berusaha untuk melakukan operasi *pre-write* data item x

if x telah dibaca atau ditulis oleh transaksi yang lebih muda seperti :

ts(Ti)< ts(read x) atau ts(Ti) < ts(write x)

then batalkan Ti dan *restart* Ti

else terima *pre-write*: *buffer* (pre) write bersama dengan ts(Ti)

end-if

end

Sejumlah operasi *pre-write* dapat tertahan di dalam buffer untuk data yang diberikan dan kebutuhan sifat *serializability* operasi *write* berhubungan dalam urutan *timestamp*. Juga ketika sebuah transaksi Ti berusaha melakukan operasi *write* di data

item x saat *commit*, ia harus mengecek apakah ada atau tidak transaksi Tj yang lebih dahulu tertahan dalam buffer menulis data item itu. Jika ada transaksi yang lebih dulu dengan nilai *timestamp* yang lebih kecil masuk misalnya Tj ditemukan, maka Ti harus menunggu hingga Tj melakukan perintah *commit* atau perintah *restart*.

Algoritma write operasi menggunakan basic timestamp

Begin

Ti berusaha untuk meng-*update* (*write*) data item x

if terdapat sebuah *update* tertahan pada data item x oleh transaksi yang lebih dahulu Tj

(misalnya: untuk ts(Tj)<ts(Ti))

then Ti menunggu hingga Tj diberi perintah *commit* atau *restart*

else Ti melakukan perintah *commit update* dan ubah ts (write x)=ts(Ti)

end if

end

Sama seperti dengan perintah read pada transaksi Ti di data item x, sistem harus tidak hanya mencek data yang telah diupdate oleh transaksi yang lebih muda, tetapi disana juga tidak ada operasi write tertahan dalam buffer untuk item data oleh transaksi lain , Ti yang lebih dulu.

Algoritma Read operation menggunakan basic timestamping

Begin

Ti berusaha melakukan operasi *read* data item x

if x telah pernah di*update* oleh transaksi yang lebih muda

(misalnya: ts(Ti)< *write*(x))

then batalkan operasi *read* dan *restart* Tj

else **if** disana sebuah *update* ditahan di x oleh transaksi yang lebih dahulu Tj

(misalnya: ts(Tj) < ts(Ti))

then Ti menunggu Tj untuk *commit* atau *restart*

else terima operasi *read* dan

set ts(*read* x)= max(ts(*read* x),ts(Ti))

end-if

end-if

end

Jika operasi *read* oleh Ti diterima, maka ts(*read* x) akan di*update* ke ts(Ti) dan operasi *write* Tj akan tidak valid ketika data item x telah dibaca oleh sebuah transaksi lebih muda Ti. Juga, dalam kasus operasi write, Ti harus menunggu hingga Tj memberi perintah commit atau restart. Ini sama dengan menggunakan *eksklusive lock* pada data item diantara operasi *pre-write* dan *write*. Ketika semua transaksi-transaksi yang lebih muda menunggu transaksi yang lebih dahulu, kemungkinan untuk terjadi *deadlock* tidak akan mungkin terjadi.

3.4.3. Dua Fase Commit

Ketika sebuah transaksi di *update*, proses tersebut menggunakan data pada beberapa *site*, ia tidak akan melakukan proses *commit* sampai yakin bahwa effect dari transaksi di setiap *site* tidak hilang. Ini berarti setiap *site* harus mempunyai catatan efek lokal dari transaksi yang secara permanen disimpan dalam lokal *site* menggunakan data log di disk. Protokol dua fase *commit* (*two-phase commit*) digunakan untuk menjamin kebenaran dari *commit* terdistribusi. Global transaksi mempunyai sebuah *site* yang bertindak sebagai **koordinator** untuk transaksi tersebut. Transaksi global mempunyai *site-site* agen yang dinamakan **partisipan**, koordinator harus mengetahui setiap partisipan yang ada.

Metoda dua fase commit beroperasi dalam dua fase : fase *voting* dan fase

keputusan. Ide dasarnya adalah semua partisipan ditanya oleh koordinator siap atau tidak mereka untuk melakukan perintah commit transaksi. Jika partisipan memilih untuk membatalkan, atau gagal untuk meresponi dalam periode waktu tunggu yang disediakan, kemudian koordinator menginstruksikan semua partisipan untuk membatalkan transaksi. Jika semua memilih untuk *commit*, maka semua partisipan diberitahu untuk melakukan perintah *commit* transaksi. Protokol ini mengasumsikan setiap *site* mempunyai lokal log dan dapat melakukan operasi *rollback* atau *commit* transaksi.

Aturan voting sebagai berikut :

(1) Setiap partisipan mempunyai satu suara yang akan menentukan '*commit*' atau 'batal'

(2) Setelah melakukan *voting* sebuah partisipan tidak dapat merubah keputusannya

(3) Jika sebuah partisipan memberikan keputusan 'batal' maka ia bebas untuk membatalkan transaksi secepatnya.

(4) Jika sebuah partisipan memilih '*commit*', maka ia harus menunggu koordintor mengirimkan pesan '*global-commit*' atau '*global-batal*'.

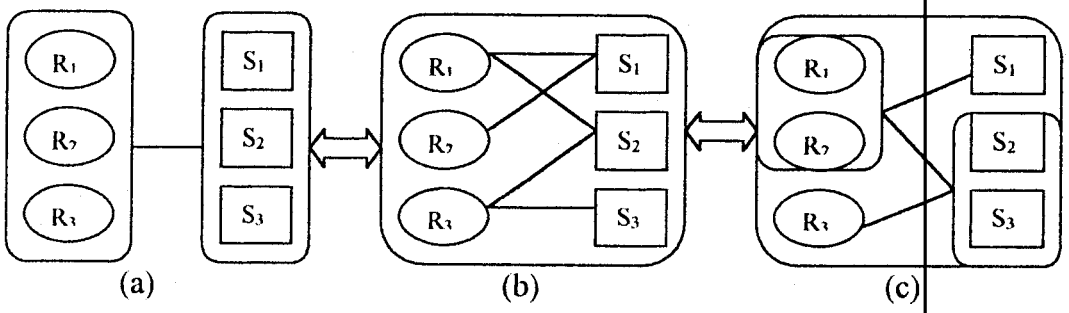
(5) Jika semua partisipan memilih '*commit*' maka keputusan koordinator secara global harus dilakukan operasi '*commit*'

(6) Keputusan global harus dilakukan di semua partisipan

3.5. Pengelompokan Proses Query

Problem pada DDBMS timbul ketika dari sebuah *join* tunggal dilakukan pada relasi-relasi terfragmentasi, dimana perlu dilakukan beberapa operasi *join* dan *union*

dengan urutan tertentu. Strategi yang dinyatakan untuk melakukan sebuah operasi *union* adalah sesuatu yang mudah, karena tidak perlu mengetahui urutan tupel-tupel dari operand yang dikoleksi dalam relasi hasil. Maka, transmisi dari non lokal relasi-relasi yang terfragmentasi ke *site-site* dimana operasi *union* dilakukan dapat dilakukan secara paralel. Dalam arti kebutuhan transmisi, operasi *union* mempunyai delay pada transmisi di *fragment* relasi terbesar, dan harga yang diberikan dari penggabungan semua harga transmisi.



Gambar 3.1 Operasi join dan union pada relasi-relasi terfragmentasi

Pada gambar 3.1. terdapat tiga gambar berbeda yang menyatakan tiga graph optimasi berbeda untuk query yang sama. Di gambar tersebut terdapat relasi terfragmentasi R yang dinyatakan dalam lingkaran, sedang relasi terfragmentasi S yang dinyatakan dalam bujursangkar. Pada dua graph optimasi pertama gambar 3.1a dan 3.1b, dimana gambar 3.1a menyatakan *fragment-fragment* yang ada dikoleksi dulu lalu baru dilakukan operasi join, dalam gambar 4.1b *fragment-fragment* dilakukan operasi join lalu baru dikoleksi. Dua kasus tersebut memberikan cara optimasi yang berbeda:

Non join terdistribusi : Metode optimasi ini sangat sederhana, ia mengurangi pasangan *site-site* yang ada (mungkin pada *site* yang sama) dimana operasi *union* dilakukan. Jika pada dua *site* yang berbeda, lalu query direduksi ke query *join* sederhana

diantara kedua relasi, yang dioptimasi di query biasa.

Join terdistribusi : Metoda optimasi ini lebih susah. Dalam gambar 3.1b ditemukan graph *join* dari *join* diantara R dan S dari garis yang ada yang menyatakan operasi *union*. Pengetahuan dari kriteria fragmentasi harus digunakan untuk mengeliminasinya dari graph *join* yang berhubungan dengan operasi *join* yang kosong. Sedang pada gambar 3.1c menyatakan kemungkinan untuk melakukan operasi union sebagian yang melibatkan beberapa *fragment* dari relasi yang sama.

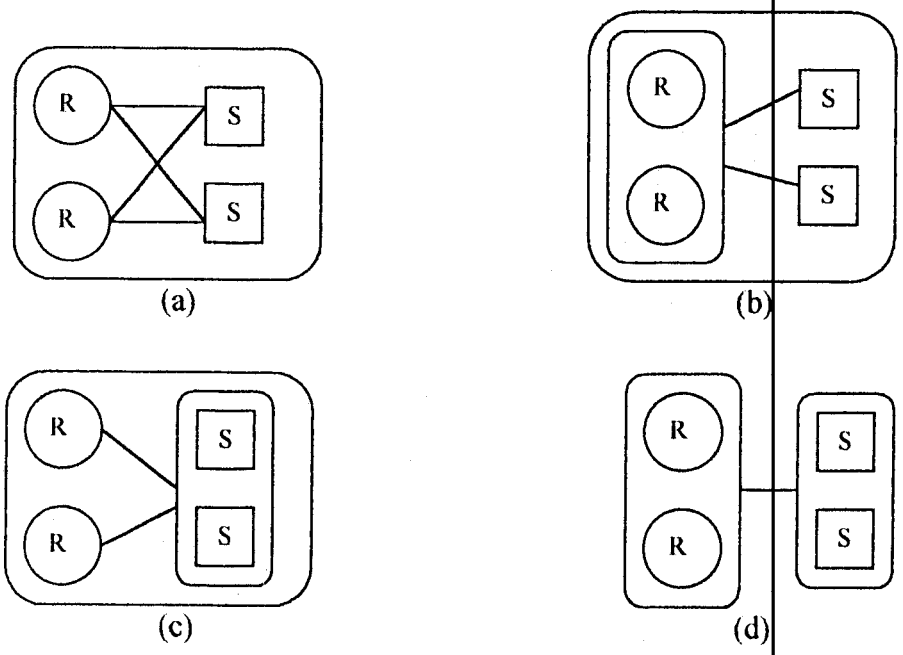
Cara berikut untuk membuat beberapa strategi yang melibatkan operasi union sebagian untuk setiap partisi dalam graph join, untuk mencari strategi eksekusi terbaik dari query yang diberikan :

- Buat semua kemungkinan graph optimasi query.
- Lakukan metode query join untuk mengoptimasi operasi join, dan menambah biaya dari operasi union, hingga setiap graph query diberikan sebagai sebuah strategi query yang optimal dan sebuah biaya.
- Pilih strategi proses query yang terbaik dari salah satu yang diatas.

Untuk menyederhanakan dapat disimpulkan empat cara alternatif untuk menghitung biaya dari graph *join* terkoneksi lengkap dari dua relasi R dan S, masing-masing mempunyai dua buah *fragment*.

1. Lakukan operasi join terdistribusi. Cara ini mengurangi perlakuan empat join dalam gambar 3.2a dan kumpulkan hasilnya. Optimasi dari keempat join tidak dapat dilakukan terpisah, proses reduksi dalam ukuran jumlah dari salah satu *fragment* dapat berguna ketika dilakukan lebih dari satu operasi join.

2. Lakukan koleksi sebagian dari *fragment* dari R, cara mereduksi untuk dua operasi join, seperti gambar 3.2b.
3. Lakukan koleksi sebagian dari *fragment* dari S, cara mereduksi untuk dua operasi join, seperti gambar 3.2c.
4. Lakukan operasi union dari *fragment-fragment*, diikuti oleh operasi join yang berhubungan dengan eksekusi tidak terdistribusi dari operasi join seperti gambar 3.2d.



Gambar 3.2 Graph optimasi altenatif untuk query yang sama

3.6. Keamanan Data

Dalam sebuah DDBMS dengan sifat autonomi pada cabang-cabangnya, keamanan data akan menjadi tanggungjawab sepenuhnya dari DBMS lokal. Pemakai dari jarak jauh memerlukan ijin untuk mengakses data lokal, *site* lokal belum tentu menjamin

keamanan dari data tersebut. Ini dikarenakan beberapa akses yang timbul selama transmisi data melalui jaringan. Tingkat keamanan secara relatif tergantung dari keamanan *site* penerimaan dan keamanan dari jaringan saat masuk ke dalam account. Terdapat beberapa kemungkinan kelemahan di titik-titik keamanan yaitu dari sebuah *site* sendiri, yaitu saat mengirimkan data penting melalui saluran komunikasi yang tidak aman atau mengirimnya ke *site* yang tidak aman. Dibagian keamanan ini ada beberapa metode yang perlu diperhatikan :

Identifikasi dan authentifikasi, ketika pemakai akan mengakses sistem komputer pertama kali, mereka harus mengidentifikasi diri mereka, dan melakukan authentifikasi indentifikasi mereka, biasanya dengan nama dan password. Pemakai dalam DDBMS diijinkan untuk mengakses data di *site* yang jauh, yang perlu untuk menaruh nama dan password mereka di semua *site*. Duplikasi informasi identifikasi dapat mengakibatkan resiko keamanan, meskipun disimpan dalam bentuk terenkripsi. Untuk menghindari duplikasi, memerlukan pemakai untuuk mengidentifikasi diri mereka sendiri di salah satu *site*, dinamakan **site rumah**, dan untuk *site* tersebut dilakukan proses authentifikasi. Sekali pemakai diijinkan oleh DDBMS di *site* mereka, semua *site-site* yang lain akan menerima user sebagai pemakai khusus. Ini tidak berarti pemakai mempunyai akses tidak terbatas di dalam DDB, mereka tetap harus memenuhi akses kontrol dari data.

Enkripsi data, enkripsi data digunakan untuk memecahkan masalah dari orang-orang yang berusaha melewati kontrol keamanan dari DBMS dan akan melakukan akses langsung ke basis data atau saluran komunikasi. Penggunaannya terutama untuk enkripsi data password, tetapi untuk data dan message dapat pula dilakukan. Data asli atau

message, dinamakan ***plaintext***, yang merupakan masukan ke algoritma enkripsi, untuk diacak dan hasilnya dalam text teracak, dengan menggunakan kunci enkripsi. Beberapa metoda enkripsi standar yang terkenal ialah *Data Encryption Standard* (DES) dan *public key crypto system.*

BAB IV

JAVA DATABASE CONNECTIVITY

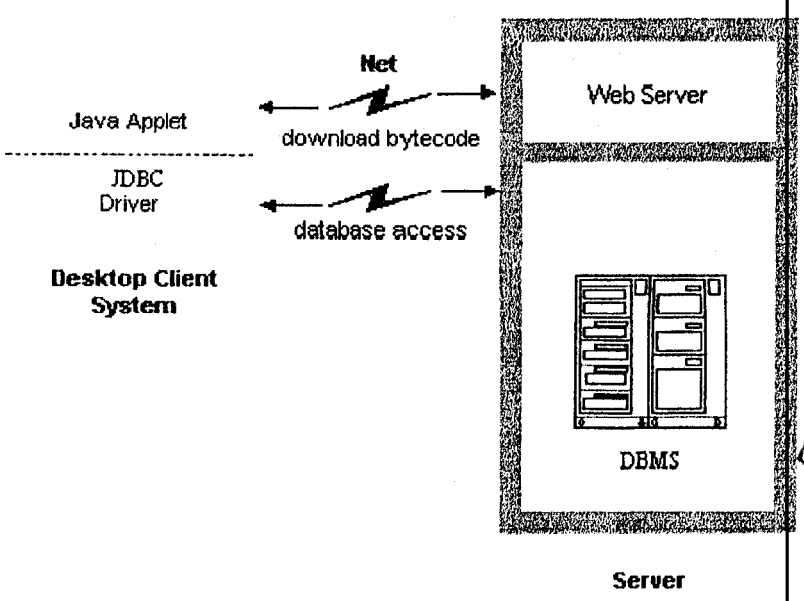
Java DataBase Connectivity (JDBC) merupakan suatu *interface* standar untuk mengakses suatu DBMS dalam lingkungan bahasa pemprograman Java. JDBC dibuat untuk memudahkan mengirim *statement query* SQL (*Structured Query Language*) ke sistem manajemen basis data relasional (RDBMS) dengan menggunakan fungsi-fungsi standar Java Application Programming Interface (API) untuk mengeksekusi *statement query* SQL tersebut. JDBC terdiri dari berbagai *class* dan *interface* yang ditulis dalam bahasa Java.

Dengan JDBC API tidak perlu menulis program yang berbeda-beda untuk mengakses basis data Sybase, program lain untuk mengakses basis data Oracle, sedangkan program yang lain mengakses basis data Informix, dan sebagainya. Cukup sekali menulis sebuah program dengan JDBC API, dan program akan mampu mengirimkan statement SQL ke basis data yang sesuai. Dengan menulis aplikasi dalam bahasa pemprograman Java, seseorang tidak perlu menulis ulang aplikasi di *platform* yang berbeda. Kombinasi Java dan JDBC membuat seorang pemprogram menulis program sekali saja dan dapat menjalankannya dimana saja. Jika digunakan dalam halaman *web* yang menggunakan *applet* yang diambil dari basis data yang jaraknya jauh, ia dapat dioperasikan melalui *internet*.

JDBC merupakan sebuah *interface* "level-rendah", yang berarti berhubungan dengan perintah SQL secara langsung. Ia mempermudah pemakaian

daripada penggunaan konektivitas API basis data yang lain. Ia didesain juga menjadi bagian dasar untuk membangun *interface* level yang lebih tinggi dan alat bantu (*tool*) yang lainnya.

Saat ini SQL merupakan bahasa standar untuk mengakses basis data relasional, Karena itu JDBC API digunakan sebagai API dasar untuk membangun akses basis data level tinggi seperti *tool* dan API yang lain. Desain "*JDBC COMPLIANT*" dibuat sebagai level standar dari fungsi-fungsi JDBC. Untuk desain ini, *driver* harus mendukung paling sedikit ANSI SQL-2 level entry (ANSI SQL-2 mereferensi ke standar yang diadopsi oleh *American National Standar Institute* dalam tahun 1992, *Entry* level mereferensi ke daftar khusus dari kemampuan SQL).



Gambar 4.1. Akses basis data jarak jauh dengan Java

Untuk akses basis data jarak jauh (gambar 4.1), *driver* JDBC harus berada di setiap sistem *client* (*web server* secara langsung memanggil *class* JDBC melalui program *applet*). Disini tidak perlu mangatur konfigurasi basis data pada aplikasi

lokal, yang akan mempermudah administrasi dan perawatan dari pemakai aplikasi. *Driver* JDBC yang digunakan sebagai bagian dari *applet* Java, dapat secara dinamik di*update* secara otomatis, sehingga mengurangi biaya pembuatan dan perawatan aplikasi. Karena JDBC sepenuhnya menggunakan bahasa Java, ia juga dapat digunakan di dalam NC (*Network Computer*) milik Sun yang baru, sebuah mesin *desktop* yang menjalankan *applet*. NC tidak memerlukan *hard drive* atau CD ROM; pemakai mengakses semua aplikasi dan data melalui *applet* yang di*download* dari *server*.

4.1. Perbandingan JDBC dengan ODBC dan API yang lainnya.

Sekarang ini Microsoft ODBC (*Open Basis data Connectivity*) API mungkin merupakan yang paling digunakan secara luas sebagai *interface* pemprograman untuk mengakses basis data relasional. Ia memberikan kemampuan untuk melakukan koneksi ke hampir semua basis data di hampir semua *platform*. Tetapi jika dibandingkan mengimplementasikan ODBC dengan Java, akan lebih baik dilakukan dengan JDBC dalam bentuk *JDBC-ODBC Bridge*, karena :

1. ODBC tidak dapat digunakan secara langsung dengan Java sebab ia menggunakan *interface* bahasa C. Pemanggilan dari Java ke *code* C *native* mempunyai sejumlah masalah dalam hal keamanan, implementasi, dan portabilitas aplikasi secara otomatis.
2. Translasi literal dari ODBC C API ke Java API akan tidak disesuai. Sebagai contoh, Java tidak mempunyai *pointer*, sedangkan ODBC memuat beberapa

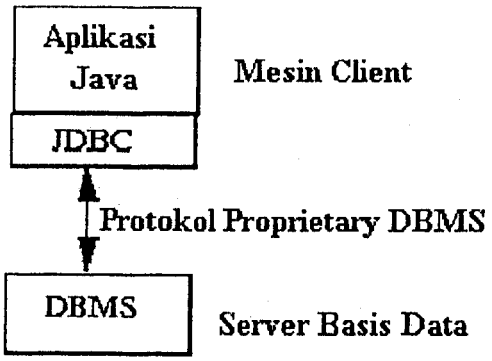
penggunaan *pointer*, termasuk penggunaannya yang seringkali mengakibatkan kesalahan pada penggunaan *pointer* murni "void *". JDBC-ODBC ditranslasi ke interface *object-oriented* yang secara alami untuk *programmer* Java.

3. ODBC seringkali susah untuk dipelajari. Ia mencampurkan ciri-ciri sederhana dan tingkat lanjut secara bersamaan, dan ia mempunyai pilihan yang komplek hanya untuk query sederhana. JDBC, dilain pihak, didesain untuk membuat segala sesuatu tetap mudah tetapi tetap mempunyai kemampuan sesuai yang dibutuhkan.
4. Sebuah Java API seperti JDBC diperlukan untuk membuat solusi "Java murni" ("*pure Java*"). Ketika ODBC digunakan, ODBC *driver* ditulis secara lengkap dalam Java, kode JDBC secara automatis dapat dipasang ulang, *portable*, dan keamanan di semua *platform* Java dari jaringan komputer sampai ke *mainframe*.

Secara ringkas, JDBC API merupakan *interface* dalam bahasa Java alami ke abstraksi dan konsep SQL. Ia dibangun berdasarkan ODBC sebagai langkah awal, sehingga pemprogram yang telah mengenal ODBC akan dengan sangat mudah mempelajari JDBC. JDBC memberikan ciri desain dasar dari ODBC yang pada kenyataannya kedua *interface* tersebut didasarkan pada X/Open SQL CLI (*Call Level Interface*). Perbedaan terbesar ialah JDBC dibangun dan digunakan dalam gaya bahasa pemprograman Java yang lebih mempermudah dalam pemakaian.

4.2. Model Two-tier dan Three-tier

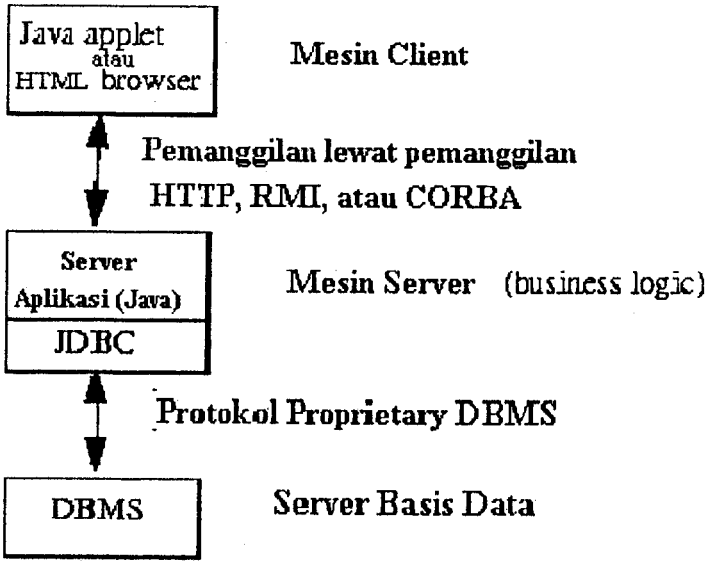
JDBC API support kedua model *two-tier* dan *three-tier* untuk akses basis data. Pada model *two-tier* (gambar 4.2), sebuah *applet* atau aplikasi Java berhubungan langsung ke basis data. Ini membutuhkan *driver* JDBC yang dapat berkomunikasi dengan sistem manajemen basis data khusus yang akan diakses. *Statement* SQL pemakai dikirimkan ke basis data, dan hasil dari *statement* tersebut dikirimkan balik ke pemakai. Basis data dilokasikan ke mesin yang lainnya dimana pemakai dihubungkan dengan jaringan. Ini merupakan tipe konfigurasi *client/server*, dimana mesin pemakai sebagai *client*, dan mesin dimana basis data berada sebagai *server*. Jaringan yang digunakan dalam model ini dapat berupa *intranet* atau *internet*.



Gambar 4.2. Model Two-tier

Dalam model *three-tier* (gambar4.3), perintah dikirimkan ke service "*middle-tier*", yang kemudian mengirimkan *statement* SQL ke basis data. Basis data memproses *statement* SQL dan mengirimkan hasil balik ke *middle-tier*, yang kemudian mengirimkannya ke pemakai. *Administrator* MIS tertarik dengan model *three-tier* sebab *middle-tier* membuat kemungkinan untuk mengatur akses kontrol dan macam-macam *update* yang dibuat untuk data bersama.

Keuntungan yang lain ialah dengan *middle-tier*, pemakai dapat menggunakan dengan mudah API level tinggi yang menterjemahkan oleh *middle-tier* ke level rendah yang sesuai. Akhirnya dalam beberapa kasus arsitektur *three-tier* dapat memberikan keuntungan kinerja. Sampai sekarang bagian *middle-tier* secara khusus ditulis dalam bahasa seperti C atau C++, dengan kinerja yang cepat. Dengan mengoptimasi compiler yang men*translate bytecode* Java ke *code* mesin spesifik yang efisien, ia akan cocok untuk mengimplementasikan *middle-tier* dalam Java, ditambah keuntungan dari Java, *multithreading*, dan keamanan.



Gambar 4.3. Model Three-tier

4.3. Tipe-tipe JDBC *driver*[9]

1. *JDBC-ODBC bridge plus ODBC driver* : Akses JDBC melalui *driver* ODBC. Catatan bahwa ODBC *binary code*, dan dalam beberapa kasus mode basis data *client*, harus dipanggil di tiap-tiap mesin *client* yang menggunakan *driver* ini. Sebagai hasil, jenis *driver* ini lebih cocok di jaringan bersama dimana instalasi

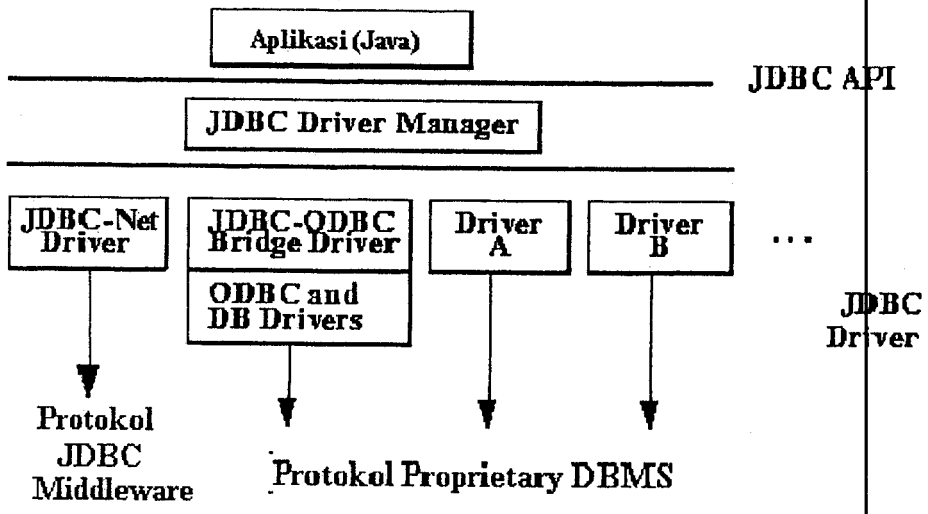
client tidak menjadi masalah utama, atau untuk *server* aplikasi yang ditulis dalam Java dalam arsitektur *three-tier*.

2. *Native-APIpartly-Java driver* : Jenis dari *driver* ini mengubah pemanggilan JDBC ke dalam pemanggilan di API *client* untuk Oracle, Sybase, Informix, DB2, atau DBMS yang lain. Catatan, untuk seperti *driver bridge*, tipe dari *driver* membutuhkan beberapa kode *binary* yang dipanggil tiap mesin *client*.
3. *JDBC-Net pure Java driver*: *driver* ini mengubah pemanggilan JDBC ke dalam DBMS protokol independen yang mengubah ke protokol DBMS oleh sebuah *server*. *Server* tengah jaringan ini mampu untuk menghubungkan *client* yang murni diprogram dalam Java ke beberapa basis data yang berbeda. Protokol khusus tergantung di *vendor*. Secara umum, ini merupakan JDBC yang paling fleksibel sebagai alternatif. Ini seperti semua *vendor* akan menyediakan produk yang sesuai untuk pemakaian *Intranet*. Dalam urutan untuk produk juga mendukung akses *Internet*, mereka harus menghandle kebutuhan tambahan untuk keamanan, akses melalui *firewall*, dan sebagainya, yang bertujuan pemakaian *web*. Beberapa *vendor* menambah *driver* JDBC ke basis data yang ada.
4. *Native-protokol pure Java driver*: Jenis dari *driver* ini mengubah pemanggilan JDBC ke protokol *network* yang digunakan oleh DBMS langsung. Ini mengijinkan pemanggilan langsung dari mesin *client* ke *server* DBMS dan sebagai solusi praktis dari akses *internet*. Sejak beberapa protokol mempunyai hak cipta sendiri, *vendor* basis data sendiri akan menjadi sumber utama, dan beberapa basis data *vendor* melakukan ini.

---

[9] "JDBC Guide: Getting Started", JDK 1.1.3, Sun Microsystems, Inc, (1997) section "Introduction"

Arsitektur tipe-tipe driver JDBC dapat dilihat pada gambar 4.4.



Gambar 4.4. Tipe-tipe Arsitektur JDBC

4.4. Penjelasan *interface* JDBC (java.sql)

JDBC API merupakan kumpulan dari *interface* java yang mengijinkan aplikasi basis data membuka koneksi ke basis data, mengeksekusi statement SQL, dan memproses hasil. Proses tersebut dalam *driver* JDBC melibatkan :

- *java.sql.DriverManager*, yang memanggil *driver* tertentu dan mendukung pembuatan koneksi ke basis data baru.
- *java.sql.Connection*, yang menampilkan sebuah koneksi ke basis data secara spesifik.
- *java.sql.Statement*, yang mengijinkan aplikasi mengesekusi pernyataan SQL.
- *java.sql.PreparedStatement*, yang menampilkan sebuah statement pre-compiled SQL.

- *java.sql.CallableStement*, yang menampilkan sebuah pemanggilan prosedur yang ditempatkan dalam basis data.
- *java.sql.ResultSet*, yang mengontrol akses ke baris dengan menyatakan statement eksekusi.

Beberapa penjelasan *class-class* dalam *driver* JDBC yang sering digunakan pada program *client* :

- Import class-class JDBC

  Class JDBC menyediakan kode ke implementasi aktual dari JDBC API. Paket java.sql mendefinisikan standar interface Java API. Paket ini mengijinkan untuk mereferensi semua class dalam interface java.sql tanpa perlu mengetahui lebih dahulu penulisan prefix java.sql. Pemakai dapat mengimport paket ini dengan baris :

  import java.sql.;*

- Class DriverManager

  Class DriverManager adalah bagian dari paket java.sql. Kerangka kerja JDBC mendukung berbagai *driver* basis data. DriverManager mengatur semua *driver* JDBC yang dipanggil oleh sistem; ia mencoba memanggil sebanyak mungkin *driver* yang dapat ia temukan. Untuk setiap permintaan koneksi, ia mengalokasikan *driver* ke untuk membuat koneksi ke URL basis data target. DriverManager juga memeriksa ukuran sekuritas yang didefinisikan oleh spesifikasi JDBC.

- Class Driver

  Setiap *driver* basis data harus menyediakan sebuah class yang mengimplementasikan interface java.sql.Driver. Class java.sql.Driver merupakan implementasi Java penuh dari *driver* JDBC yang secara khusus.

- Class Connection

  Setelah membuat objek Driver, membuka koneksi ke basis data dapat dengan membuat objek Connection. Sebuah *driver* basis data dapat mengatur beberapa objek Connection. Objek Connection mensahkan dan mengatur koneksi ke basis data secara khusus. Dengan koneksi yang diberikan dapat dieksekusi pernyataan SQL dan menerima hasil.

  Spesifikasi JDBC mengijinkan aplikasi tunggal untuk mendukung beberapa koneksi untuk satu atau lebih basis data, menggunakan satu atau lebih *driver* JDBC. Ketika mensahkan koneksi menggunakan class ini, DriverManager memilih *driver* yang sesuai dari yang dipanggil berdasarkan dari subprotocol spesifikasi di URL, yang dilewatkan ke parameter koneksi.

4.5. Cara menggunakan *driver* JDBC

Penggunaan *driver* JDBC di program client pada dasarnya ada dua cara :

- Applet Java yang dipanggil di dalam halaman html dengan tag <APPLET>, ditayangkan melalui sebuah web server, dan dilihat dan digunakan sistem client dengan web browser yang mempunyai kemampuan menggunakan Java. Penggunaan metode ini tidak membutuhkan instalasi manual dari paket *driver*

JDBC, sedang dari sisi sistem client membutuhkan browser berkemampuan Java.

- Aplikasi Java merupakan program Java yang berdiri sendiri yang dieksekusi di sistem client. Penggunaan metode ini membutuhkan instalasi paket *driver* JDBC, dan Java Runtime Environment (JRE 1.1) yang diinstall di sistem client.

Langkah-langkah penggunaan driver JDBC sebagai berikut :

- Pemanggilan *driver* JDBC

  Driver JDBC harus dipanggil sebelum aplikasi berusaha untuk memulai hubungan dengan basis data. Secara eksplisit memanggil *driver* JDBC dengan DriverManager, dimasukkan dalam baris dalam program pemakai sebelum menggunakan *driver* untuk mensahkan koneksi basis data:

  Class.forname("xxx.Driver");

  Pertama kali interpreter Java melihat referensi *xxx.Driver*, ia akan memanggil *driver xxx*. Ketika *driver* dipanggil secara automatis ia membuat dirinya sendiri. Tetapi tidak ada handle yang dapat mengakses *driver* langsung dengan nama *driver* tersebut. *Driver* ini '*anonymous*"; tidak perlu mereferensi itu secara eksplisit untuk memuat koneksi basis data. Koneksi basis data sederhana menggunakan class java.sql.DriverManager.

  Ia bertanggungjawab untuk setiap *driver* baru yang dipanggil dengan meregistrasi sendiri melalui DriverManager; pemprogram tidak perlu meregister *driver* secara eksplisit. Setelah *driver* diregistrasi, DriverManager dapat menggunakannya untuk membuat koneksi ke basis data. Contoh berikut

menunjukkan bagaimana interaksi dengan basis data dengan referensi langsung *driver*:

```
// membuat iddb driver objek sebagai driver jdbc
java.sql.Driver driver = new iddb.driver();
// ambil objek koneksi
java.sql.Connection connection = driver.connect (dbURL, properties);
// referensi driver untuk mengambil nomor versi driver
java.sql.String version=driver.getMajorVersion()+ driver.getMinorVersion();
```

- Membuka koneksi ke basis data

Setelah memanggil *driver*, untuk membuka koneksi dapat memanggil class Connection melalui motode DriverManager getConnection() atau objek driver dengan metode connect(). Contoh pemanggilan :

```
Connection c = DriverManager.getConnection (url,properties);
```

Struktur URL JDBC adalah sebagai berikut :

```
jdbc:subprotokol:subname
```

Subprotokol merupakan jenis dari koneksi basis data yang didukung oleh satu atau lebih tipe basis data. DriverManager akan memutuskan subprotokol mana yang cocok dengan *driver* basis data yang sesuai. Ini dan sintaks dari subname tergantung pada subprotokol. Perjanjian nama seringkali sebagai berikut :

```
//hostname:/subsubname
```

contoh pemanggilan :

```
jdbc:oracle://server/directory_&_nama_lengkap_db
```

- Class untuk mengeksekusi statement SQL

Disini terdapat tiga class **java.sql** untuk mengeksekusi statement SQL :

- *Statement* : eksekusi statement SQL dan pemanggilan hasil yang diproduksi oleh query.
- *PreparedStatement* : mengijinkan sekelompok statement SQL dieksekusi lebih dari sekali. Disamping membuat statement baru setiap kali di fungsi yang

sama, dapat digunakan untuk mengeksekusi statement pre-compiled SQL berulang-ulang.

- *CallableStatement* : digunakan untuk mengeksekusi prosedur yang diberikan (disimpan) dengan parameter OUT.

Contoh pembuatan objek Statement :

```
Java.sql.Statement statement = connection.createStatement();
```

Untuk proses query data, jika melakukan operasi pembacaan (operasi SELECT) menggunakan metode executeQuery yang akan menggembalikan objek ResultSet yang menggunakan kumpulan metode *get* yang membuat pemakai dapat mengakses kolom dari baris saat itu. ResultSet.next untuk berpindah ke baris selanjutnya.

Contoh :

```
//Eksekusi statement SELECTdan menaruh hasil di ResultSet:
java.sql.ResultSet resultSet = statement.executeQuery
("SELECT nama, tgllahir, no
   FROM pegawai_table
   WHERE dept_nama = 'accounting'");

//Tampilkan baris hasil
system.out.println("Dapat hasil:");

while (resultSet.next ()){

  //ambil nilai dari baris sekarang
  string nama = resultSet.getString(1);
  string tgllahir = resultSet.getString(2);
  string no = resultSet.getString(3);

  //tampilkan hasil
  system.out.print(" Nama=" + nama);
  system.out.print(" Tgllahir=" + tgllahir);
  system.out.print(" Nomor=" + no);
  system.out.print("\n");
}
```

Sedangkan untuk melakukan *statement* operasi *update* (INSERT, DELETE, UPDATE) pada *statement query* digunakan metode executeUpdate, yang akan mengembalikan jumlah baris yang diubah oleh operasi tersebut.

Contoh :

```
int rowCount = statement.executeUpdate("INSERT INTO nama_table VALUES( val1, val2,...)");
```

BAB V

PERANCANGAN DAN PEMBUATAN PERANGKAT LUNAK

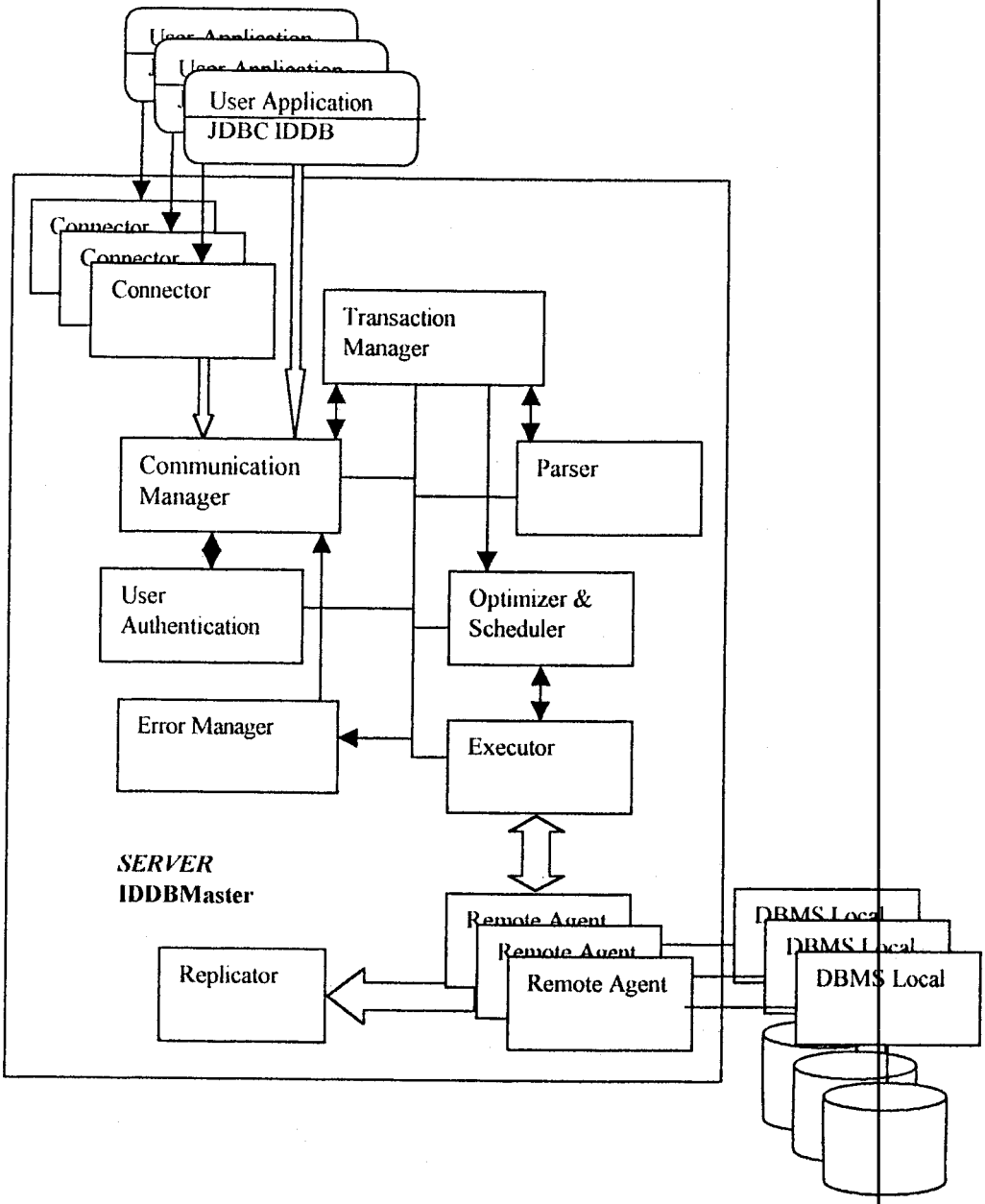
Dalam bab ini dijelaskan mengenai perancangan dan pembuatan perangkat lunak prototipe sistem manajemen basis data terdistribusi, dengan menggunakan dasar teori yang telah dijelaskan pada bab sebelumnya. Hal yang dibahas meliputi arsitektur, proses, dan implementasi.

5.1. Arsitektur Prototipe Sistem Basis Data Terdistribusi

Perancangan sistem merupakan tahap untuk mentransformasikan berbagai algoritma sistem manajemen basis data terdistribusi ke dalam arsitektur program. Hal ini meliputi pendefinisian modul-modul yang ada dan operasi yang dilakukan dalam modul-modul tersebut.

Prototipe sistem manjemen basis data terdistribusi ini dinamakan ITS Distributed DataBase (IDDB). Sistem ini dirancang untuk melayani permintaan fragmentasi data secara horisontal dengan aturan fragmentasi menggunakan parameter *field-field* pada tabel itu sendiri, ia tidak dapat mengambil *field-field* dari tabel lain saat pengecekan aturan fragmentasi untuk perintah *query* insert. Tiap-tiap DBMS lokal mempunyai jumlah tabel yang sama dengan struktur data yang sama pula, sistem IDDB ini tidak mengijinkan hanya sebagian tabel saja yang ditempatkan di salah satu DBMS lokal.

Pembagian modul-modul dalam sistem basis data ini (gambar 5.1) secara



Gambar 5.1. Arsitektur IDDB

singkat sebagai berikut :

- **Aplikasi *Client*** : merupakan aplikasi sistem informasi yang digunakan oleh pemakai, ia mengirimkan perintah kepada *server* basis data melalui *driver* JDBC, menerima hasilnya, dan mengolah hasil tampilan itu di layar program aplikasi.
- **JDBC IDDB** : merupakan *interface* yang menghubungkan program aplikasi dengan *server*. Disini dia berfungsi sebagai penghubung komunikasi dalam jaringan komputer. Dengan digunakannya standar JDBC sebagai *interface*, maka pemakai

dapat dengan mudah mengganti *driver* JDBC ini dengan *driver* JDBC yang lain tanpa tersebut sesuai dengan kebutuhannya untuk menggunakan suatu DBMS tertentu, tanpa harus meng*compile* ulang program aplikasi.

- **Communication Manager** : merupakan bagian yang mengatur lalulintas komunikasi antar *client* dengan *server*. Dia berjaga-jaga dengan menunggu saluran komunikasi *server* pada port tertentu, kalau ada *client* yang masuk, ia membuatkan suatu bagian connector yang baru, serta menyerahkan saluran komunikasi selanjutnya ke bagian connector terebut. Disini saat pertama kali berkomunikasi pemakai diperiksa identifikasinya serta *password* dengan berhubungan dengan bagian User Authentication untuk mensahkan atau menolak hubungan koneksi. Tugasnya yang lain meneruskan pesan / data dari bagian yang lain ke bagian connector yang sesuai
- **Connector** : merupakan bagian yang memelihara hubungan dengan *client* secara langsung. Jumlahnya sebanyak jumlah *client* yang terkoneksi saat itu. Segala perintah dan hasil yang ada berhubungan langsung dengan *client* harus melewati bagian ini. Ia merupakan sebuah class *thread* yang harus selalu berjaga-jaga atas kedatangan perintah dan data yang ada.
- **User Authentication** : merupakan bagian yang pengecekan apakah identitas pemakai ada dalam basis data pemakai serta validitas *password* pemakai.
- **Error Manager** : merupakan bagian yang mengolah *output error* yang ada dan memberikan hasilnya ke communication manager untuk disampaikan ke *client*, serta menampilkan ke layar atau ke *file* log bila perlu.
- **Transaction manager** : merupakan bagian yang mengatur masalah *concurrency control*, dimana transaksi-transaksi yang ada dapat dijamin berjalan bersamaan tanpa saling mengganggu satu dengan yang lainnya. Ia yang memutuskan apakah transaksi

yang melayangkan permintaan "commit" diterima atau harus ditolak. Disini mengimplementasikan algoritma timestamp.

- **Sql parser** : merupakan bagian yang mengolah permintaan *query* dari aplikasi *client*, yang berupa perintah SQL, dicek kebenaran sintaksnya dan diubah menjadi struktur data yang dimengerti oleh bagian selanjutnya agar mudah untuk diolah.
- **Optimizer & scheduler**: merupakan bagian yang mengolah hasil struktur data hasil parser tersebut menjadi langkah-langkah *query* dalam urutan sesuai urutan yang akan dieksekusi ke semua basis data yang ada pada *site-site* yang lain, yang telah dioptimasi untuk mempercepat prosesnya,. Algoritma yang digunakan dalam bagian ini menggunakan operasi semi-join (untuk perintah select).
- **Executor** : merupakan bagian yang melaksanakan rencana langkah-langkah *query* yang dihasilkan oleh modul diatasnya. Ia mengolah *query-query* tersebut, memberikannya ke semua *site-site* yang diperlukan melalui bagian remote agent. Ia mengolah juga semua hasil yang diberikan oleh semua remote agent, menggabungkannya dan memberikan hasil ke modul diatasnya.
- **Remote Agent** : merupakan bagian yang mengolah langkah-langkah *query* yang diberikan pada sebuah *site* lokal tertentu. Ia merupakan sebuah class *thread* yang jumlahnya sebanyak *site-site* dalam fragmentasi basis data.
- **Replicator** : merupakan bagian yang mengatur masalah replikasi data.
- **DBMS Lokal** : merupakan sistem manajemen basis data yang ada pada tiap-tiap *site* yang tersebar dalam jaringan komputer, yang terfragmentasi menurut aturan tertentu, serta menangani pengaksesan data secara lokal.

5.1.1. Penjelasan Modul-Modul Dalam IDDB

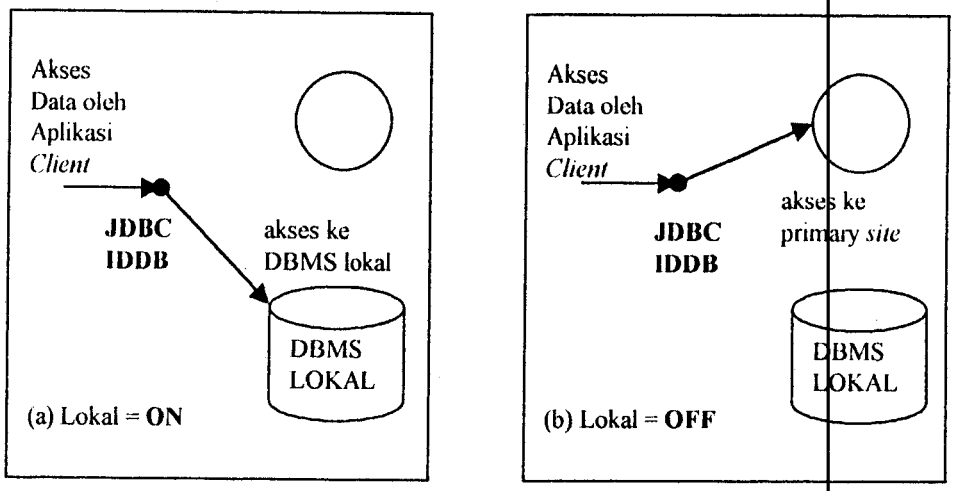
Pada bagian ini akan dijelaskan secara detail tiap-tiap modul yang ada pada prototipe sistem manajemen basis data terdistribusi IDDB.

5.1.1.1. JDBC IDDB

Dalam *driver* JDBC ini, ia berfungsi sebagai *interface* dari program aplikasi pemakai ke *IDDBMaster*. Dimana ia menghubungkan semua perintah-perintah *query* yang diberikan kepadanya ke *IDDBMaster* dan mengirimkan hasilnya balik ke program aplikasi, selain itu ia menerima pesan-pesan *error* atau *warning* dari *server IDDBMaster* ke program aplikasi.

Bagian *driver JDBC IDDB* ini mengimplementasikan arsitektur Sun JDBC standar sesuai dengan bab IV, dengan penyesuaian protokol komunikasi sesuai pada subbab penjelasan protokol komunikasi berikut dalam bab ini. JDBC API merupakan kerangka kerja dari IDDB API, dimana IDDB API sebagai *interface* aplikasi program Java untuk sistem IDDB ini, dengan menggunakan bahasa SQL untuk perintah-perintah *query*.

Untuk pengaksesan data secara lokal, maupun global, diimplementasikan keduanya dalam *driver* ini. Dalam proses pengaksesan lokal atau global, ia seperti saklar yang akan memindahkan akses data ke DBMS lokal atau ke *primary site* (*IDDBMaster*), hal tersebut dapat dilihat di gambar 5.2 (gambar 5.2a) saat pilihan lokal = *on* akses ke DBMS lokal, sedang saat pilihan lokal = *off* akses ke *primary site* (gambar 5.2b). Saat akses ke DBMS lokal bedanya dengan menggunakan *driver*



Gambar 5.2. Akses data melalui JDBC IDDB oleh aplikasi *Client*

DBMS lokal tersebut secara langsung ialah *driver JDBC IDDB* ini melakukan *update* terhadap tabel-tabel yang merupakan bagian dari database IDDB saat dilakukan proses *query* yang melakukan *update* data.

Database IDDB menggunakan tabel log yang mencatat semua *update* yang dilakukan pada data DBMS lokal, gunanya, saat dilakukan proses sinkrosisasi oleh modul *Replicator*, dapat diketahui mana data yang baru dimasukkan atau yang baru di*update*. Untuk melakukan hal itu dilakukan proses parsing sederhana guna menentukan jenis perintah *query update* (insert, update, delete) yang harus dimasukkan ke tabel log atau hanya perintah baca (select) saja, jika diketahui perintah *query* tersebut merupakan *query update*, maka akan dimasukkan catatan perubahan tersebut ke tabel log masing-masing sesuai nama tabel yang dikenai proses *query update* tersebut, lalu meng*update* data nomor urut perubahan tabel tersebut pada tabel *iddbtbcount*. Tabel log tersebut mempunyai nama **log***nama_tabel*.

Sesuai dengan spesifikasi standar JDBC, transaksi digabung menjadi satu ke

sebuah Connection. Sehingga di suatu saat, setiap koneksi dianggap sebagai sebuah transaksi yang berhubungan dengan IDDB. JDBC menspesifikasikan bahawa transaksi tidak akan *auto-commit* sampai *statement* dipakai kembali atau ditutup.

5.1.1.2. Communication Manager

Dalam modul ini, tugasnya menunggu aplikasi *client* yang menggunakan *IDDB JDBC*, yang akan masuk ke *server IDDBMaster*, jika ada yang masuk, maka akan dibuat bagian *Connector* yang baru, selanjutnya proses komunikasi dilayani oleh bagian *Connector*. Selain itu ia berisi daftar kumpulan semua connector yang masih aktif, sehingga kalau diperlukan dapat menghubungkan pesan atau hasil proses *server* ke *Connector* yang sesuai, yang akan diteruskan *Connector* ke aplikasi *client*.

5.1.1.3. Connector

Dalam modul ini ia berhubungan langsung ke *driver JDBC IDDB* yang digunakan oleh program aplikasi *client*, proses dalam bagian ini mulai diadakan ketika socket hubungan komunikasi pertama kali diberikan oleh *Communication Manager*. Protokol komunikasi pada subbab berikut diimplementasikan di bagian ini, sebagai kebalikan dari sisi pengirim *JDBC IDDB*.

Pertama-tama diterima data-data inisialisasi seperti : url, user, dan password. Jika data user dan password dikenali oleh modul *User Authentication* maka koneksi dapat terus dilanjutkan (disahkan) kalau tidak keluar pesan error dan koneksi diputuskan.

Disini terdapat procedure untuk mengirimkan tipe data *field-field*, serta data-

data record (tuple). Modul ini berbentuk suatu class *thread* yang akan berjalan bersama-sama dengan bagian Connector-Connector yang lainnya agar program aplikasi *client* dapat dilayani sendiri-sendiri secara paralel.

5.1.1.4. User Authentication

Bagian ini pertama-tama mengambil data user dari tabel 'iddbuser' sesuai dengan nama user yang diberikan. Disini dilakukan pembandingan apakah nama user dan password sesuai, jika sukses akan diambil juga level dari user tersebut. Sedangkan proses encrypt dan decrypt dilakukan dalam bagian ini agar kata-kata password yang dimasukkan oleh setiap user tidak dapat dilihat oleh sembarang orang.

5.1.1.5. Error Manager

Bagian ini akan menampilkan kesalahan (error) atau pesan peringatan (warning) ke layar atau ke *driver IDDB JDBC*. Untuk mengeluarkan pesan ke *driver IDDB JDBC*, ia berhubungan dengan *Communication Manager* untuk diteruskan ke *Connector* yang sesuai.

5.1.1.6. Transaction Manager

IDDB dalam prosesnya setiap saatnya hanya memproses satu perintah *query* SQL, sedangkan program aplikasi *client* dapat lebih dari satu. Agar menjaga proses perubahan data dilakukan secara serializable, maka digunakan *Transaction Manager* untuk mengatur transaksi agar jalan secara bersamaan (concurrent).

Bagian ini melakukan manajemen transaksi yang diberikan oleh tiap-tiap

program aplikasi *client* yang dibedakan menurut nomor transaksinya, disini setiap perintah *query* yang masuk diberikan nilai timestamp, lalu dilakukan operasi parsing oleh modul *Parser*, dari sini dapat diketahui apakah *query* yang diberikan tersebut berjenis operasi *update* atau read. Jika berjenis operasi *update* (insert, update, delete) maka perintah tersebut dimasukkan ke queue *update* transaksi, sedang jika *query* merupakan perintah read (select) akan segera dilakukan operasi pembacaan dan mengirimkan hasilnya ke *client*. Saat diberikan perintah 'commit', maka semua perintah yang ada pada transaksi *update* akan diproses sesuai dengan urutan *query* dalam transaksi tersebut, jika *query* tersebut tidak memenuhi concurrency control maka akan dilakukan proses 'rollback'. (Algoritma lengkap timestamp ini dapat dilihat di subbab 3.4.2)

Tipe perintah *query* selain dari select, insert, update, dan delete akan langsung dilakukan tanpa dimasukkan proses transaksi. Pada bagian ini proses penguncian (locking) dilakukan pada tabel-tabel, tipe penguncian ekslusive dilakukan jika dilakukan operasi *query* tipe *update*, sedangkan tipe penguncian sharing dilakukan jika dilakukan operasi *query* tipe read.

5.1.1.7. SQL Parser

Bagian ini digunakan untuk mengenali grammar SQL minimum (sintaks SQL minimum grammar dapat dilihat di lampiran A). Hasil dari modul ini berupa struktur data hasil interpretasi parser yang dapat dimengerti oleh bagian lain. Proses parsing secara umum, menentukan perintah *query* yang dimasukkan tersebut merupakan tipe perintah *query* apa.

5.1.1.8. Optimizer & scheduler

Bagian ini merupakan pengabungan antara bagian optimizer dan scheduler karena bagian ini belum terlalu kompleks dan untuk menghemat jumlah perpindahan data yang harus dibawa saat melaksanakan prosesnya. Pada bagian ini jika diketahui perintah yang akan dikerjakan merupakan perintah select maka akan dilakukan optimasi. Diproses menurut algoritma semijoin, (sebagai contoh hasil langkah-langkah rencana pemrosesan *query* dapat dilihat di lampiran D). Hasil output nanti berupa daftar urutan langkah-langkah proses operasi semi-join, pada struktur data schedulelist.

5.1.1.9. Executor

Melaksanakan rencana langkah-langkah *query* yang dihasilkan oleh modul diatasnya, memberikannya ke semua *site-site* yang dibutuhkan. Serta mengolah semua hasil yang diberikan oleh semua remote agent, menggabungkannya jika diperlukan dan memberikan hasil atau laporan ke modul diatasnya. Langkah-langkah yang dilakukan untuk tiap-tiap perintah *query*:

Perintah CREATE TABLE:

Disini diketahui *field-field* tabel yang akan disimpan dalam basis data iddbtb, iddbtbsite, iddbtbfield. Dicek apakah nama tabel pernah dimasukkan apa belum, kalau sudah pernah maka akan ditolak. Di iddbtb dimasukkan nama tabel serta pemilik (owner) yang diketahui saat koneksi, di iddbsite dimasukkan *site-site* yang berhubungan dengan *site-*

site yang ditunjukan sebagai perpecahan data (*fragment*), dis ni terdapat pula aturan *fragment*, di iddbtbfield, dimasukkan nama *field*, serta tipe data dan panjangnya, serta data lain yang diperlukan, Selanjutnya perlu dilakukan perintah *query*: extended table create, untuk menambah fragmentasi dan default *site*, atau ditempatkan secara default di *site* pertama.

Perintah DROP TABLE:

Semua bagian tabel yang berhubungan dengan nama tabel tersebut dihapus semua, di tabel iddbtb, iddbtbsite, iddbtbfield, serta semua tabel di *site-site* fragmentasi dihapus semua.

Perintah INSERT:

Jika type insert yang dipilih bertipe *fragment*, dari struktur data hasil proses parser akan dianalisa apakah nilai yang masukkan itu sesuai dengan salah satu aturan *fragment* yang diberikan, jika sesuai maka akan dimasukkan ke *site* dengan aturan *fragment* tersebut, jika tidak dilihat apakah ada salah satu *site* yang ditunjuk sebagai default *site*, jika ada maka akan dimasukkan ke *site* tersebut jika tidak maka akan dimasukkan ke *site* yang pertama dalam daftar *site*.

Jika digunakan tipe random, maka tuple baru tersebut akan dimasukkan secara acak ke salah satu *site* tabel tersebut, sedang jika dipilih urut, maka dipilih urutan *site* berikutnya dalam daftar *site* tabel tersebut. Jika sukses akan dikembalikan jumlah tuple hasil perintah insert.

Perintah DELETE dan UPDATE:

Query akan dikirimkan langsung ke semua *site-site* yang mempunyai tabel yang diminta. Hanya pada perintah update jika ada nilai yang akan merubah salah satu attribut aturan fragmentasi maka perintah update akan ditolak dengan dikirimkan pesan error. Jika berhasil akan dikirimkan jumlah tuple yang di*update*.

Pada bagian replikasi data, jika type replikasi saat itu syncronous, maka saat *update* dilakukan akan dilakukan perintah update pula pada semua data yang ada di data replikasi. Perintah ini diakses dengan perintah executeUpdate.

Perintah SELECT:

Perintah ini merupakan yang paling komplek dari semua perintah yang ada, jika tipe optimasi semijoin, maka pertama-tama setelah diparsing ia dilakukan operasi semijoin, yang akan menghasilkan langkah-langkah pelaksanaan *query* semijoin, yang dilaksanakan di bagian executor ini.

Jika tipe optimasi direct, maka data-data dari *site-site* lokal akan diambil dan diberikan ke program aplikasi *client*. Jika tipe optimasi replikator, maka *query* menggunakan data replikasi.

Jika tipe replikasi syncronous, maka sebelum perintah *query* select ini dilakukan akan di lakukan proses sinkronisasi antara data di lokal maupun di bagian replikasi. Perintah ini diakses dengan perintah executeQuery.

Jika hasil yang ada harus dikumpulkan dibagian result, digabung baru dikirimkan ke bagian transaksi Jika berhasil akan dikirimkan tuple-tuple hasil. Perintah ini diakses dengan perintah executeQuery.

Perintah EXTENDED TABLE CREATE

Perintah ini digunakan untuk memasukkan informasi aturan fragmentasi tabel. Disini dicek apakah tabel dan *site* sudah pernah dimasukkan belum, kalau sudah maka akan dimasukkan data aturan fragmentasi ke tabel iddbtbsite.

Perintah EXTENDED TABLE DROP

Perintah ini digunakan untuk menghapus informasi aturan fragmentasi tabel. Disini dicek apakah aturan fragmentasi sudah pernah dimasukkan atau belum, kalau pernah hapus data tersebut dari tabel iddbtbsite.

Perintah EXTENDED SITE CREATE

Perintah ini digunakan untuk memasukkan *site-site* lokal yang digunakan dalam database ini. Pertama kali dicek apakah sudah pernah dimasukkan *site* ini dalam tabel iddbsite, kalau sudah akan diberi pesan error, kalau belum akan dimasukkan ke tabel tersebut, lalu dijalankan ***Remote Agent*** yang baru yang mewakili *site* yang datanya baru dimasukkan tersebut.

Perintah EXTENDED SITE DROP

Perintah ini digunakan untuk menghapus *site-site* lokal yang telah dimasukkan dengan perintah 'extended site create'. Pertama-tama dicek

apakah *site* tersebut pernah dimasukkan, kalau belum akan diberi pesan error, sedangkan jika pernah akan di hapus dari tabel iddbsite, lalu bagian *Remote Agent* yang mewakili *site* lokal tersebut dihentikan dari memory.

5.1.1.10. Remote Agent

Bagian ini mewakili DBMS lokal yang akan diakses oleh modul-modul yang lain. Ia merupakan suatu class *Thread* sehingga dapat bekerja secara paralel jika permintaan *query* dilayangkan ke DBMS-DBMS lain, tanpa harus menunggu *Remote Agent* yang lain harus selesai dahulu.

5.1.1.11. Replicator

Bagian Replicator akan mencatat semua perubahan yang ada antara basis data replikasi dengan basis data utama di DBMS lokal, dengan membandingkan tabel log yang ada diantara kedua tabel log tersebut. Jika terdapat perbedaan, maka akan dilakukan proses *update*, dan dicatat counter yang sama bahwa saat selesai tersebut basis datanya telah sama. Disini terdapat dua macam operasi sinkronisasi yang dilakukan yaitu berjenis sinkronous atau asinkronous, jika ia bertipe asinkronous, berarti untuk setiap waktu yang tetap sesuai nilai yang dimasukkan di file IDDBMaster.ini ia akan meng*update* data. Kalau tipe sinkronous saat dilakukan operasi *query* tipe *update* dan read, maka basis data replikasi juga akan di*update* juga serta dicatat pada tabel log.

5.1.2. Penjelasan Komunikasi *Client / Server* pada IDDB

Prinsip dasar dari sistem ITS Distributed Database (IDDB) ini ialah setiap *query* yang masuk disalurkan ke DBMS-DBMS lokal yang ada (Gambar 5.3). Disini *query* dikenali tipe perintahnya, jika tipe select, update, delete, maka *query* tadi akan disebarkan langsung ke semua DBMS-DBMS lokal yang ada (Gambar 5.3a). Sedangkan jika tipe insert maka akan dipilih salah satu DBMS lokal sebagai tempat penyimpanan datanya (Gambar 5.3b).

Pemilihan DBMS lokal sebagai tempat penyimpanannya pada perintah *query* bertipe insert berbeda-beda sesuai dengan konfigurasi IDDBMaster, jika ia bertipe random, maka pemilihan akan dilakukan secara acak, sedangkan jika bertipe order, maka akan dipilih DBMS lokal yang berikutnya sesuai urutan, sedangkan jika bertipe fragment, maka ia akan mencari DBMS lokal yang paling tepat yang sesuai dengan aturan fragmentasi tabel yang sudah diberikan di bagian sebelumnya.

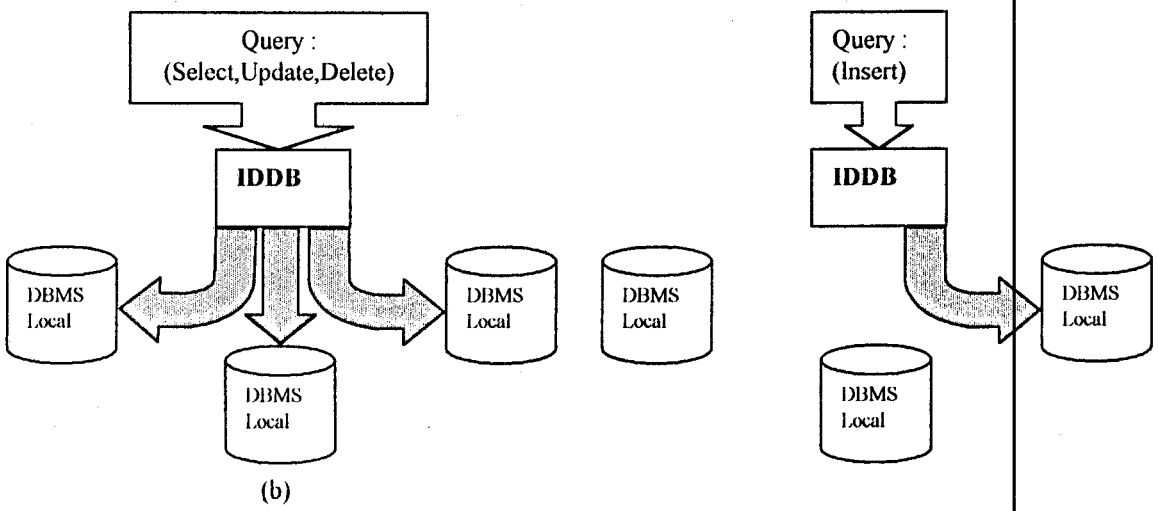
Secara global arsitektur IDDB ini dapat dilihat di Gambar 5.4, disini terlihat bahwa program-program aplikasi *client* di hubungkan ke *server IDDBMaster* melalui *JDBC IDBC*, selanjutnya dari *IDDBMaster* akan dikirimkan perintah-perintah *query* ke DBMS lokal melalui *driver* JDBC masing-masing DBMS lokal tersebut (Gambar 5.4a). Untuk Gambar 5.4b terlihat dimana sistem IDDB ini diakses secara bersamaan baik secara global maupun secara lokal. Jika ia mengakses secara lokal maka ia dapat berhubungan langsung dengan DBMS lokal melalui *driver JDBC IDDB*, sedangkan jika ia mengakses secara global maka ia akan berhubungan ke *server IDDBMaster* melalui *JDBC IDDB* pula. Pada saat pemakai mengakses data secara global, semua

perubahan yang dilakukan secara lokal akan tampak secara transparan oleh pemakai IDDB yang mengakses data secara global.

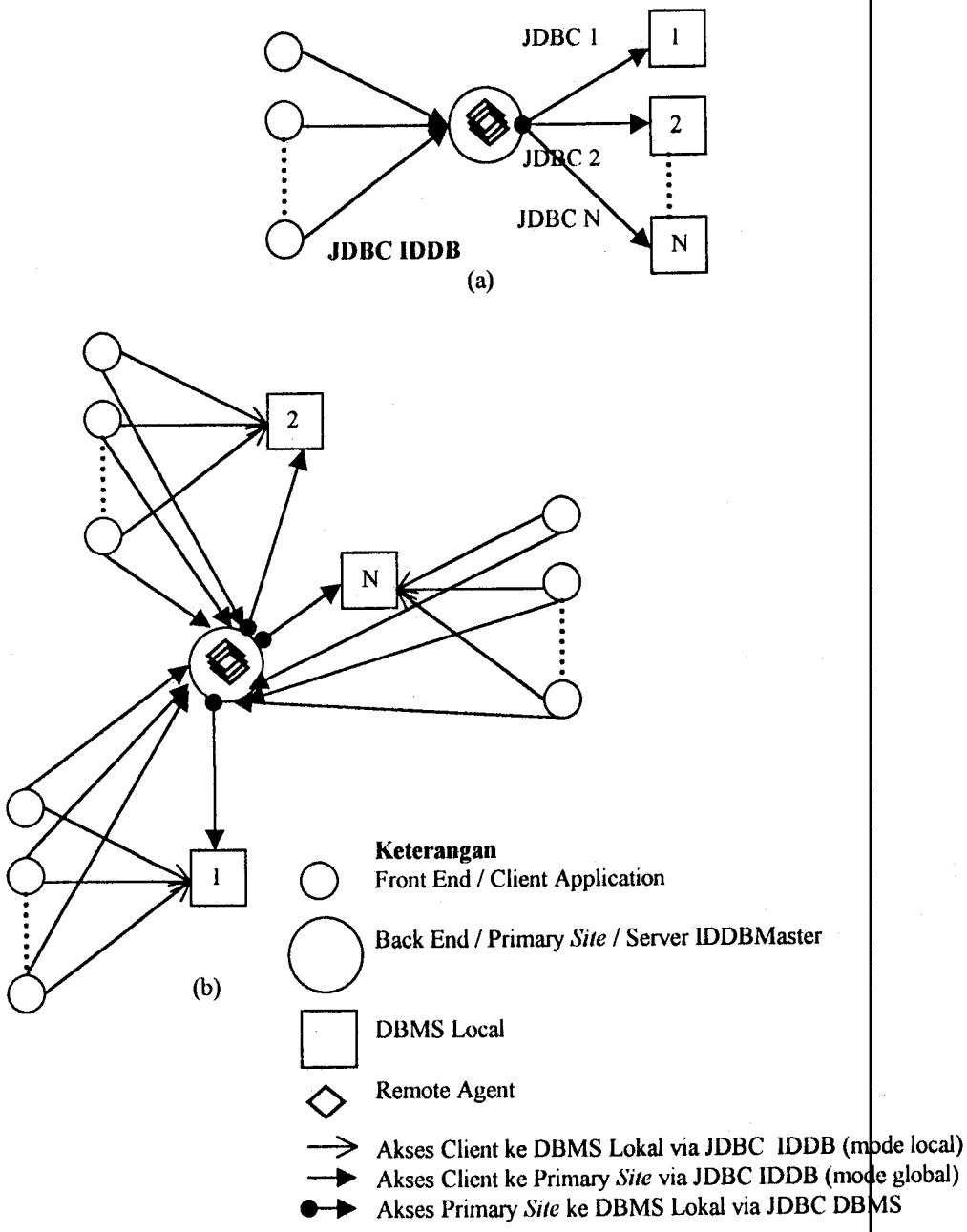
Sedangkan untuk perintah *query* bertipe select penyaluran perintah *query*nya dapat berbeda-beda pula sesuai dengan konfigurasi *IDDBMaster*, jika ia bertipe direct maka perintah *query* akan disalurkan langsung ke semua DBMS lokal, sedangkan jika bertipe replicator, perintah *query* diambil dari data replikasi, sehingga *query* bisa dilakukan lebih cepat dan beban komunikasi dikurangi, sedangkan jika bertipe semijoin, perintah *query* dilaksanakan menurut langkah-langkah sesuai algoritma semijoin.

Sistem basis data terdistribusi ini menggunakan arsitektur model *client/server* dengan metode "*connection-per-user*" sederhana. Sebuah *session* IDDB terdiri dari proses berikut :

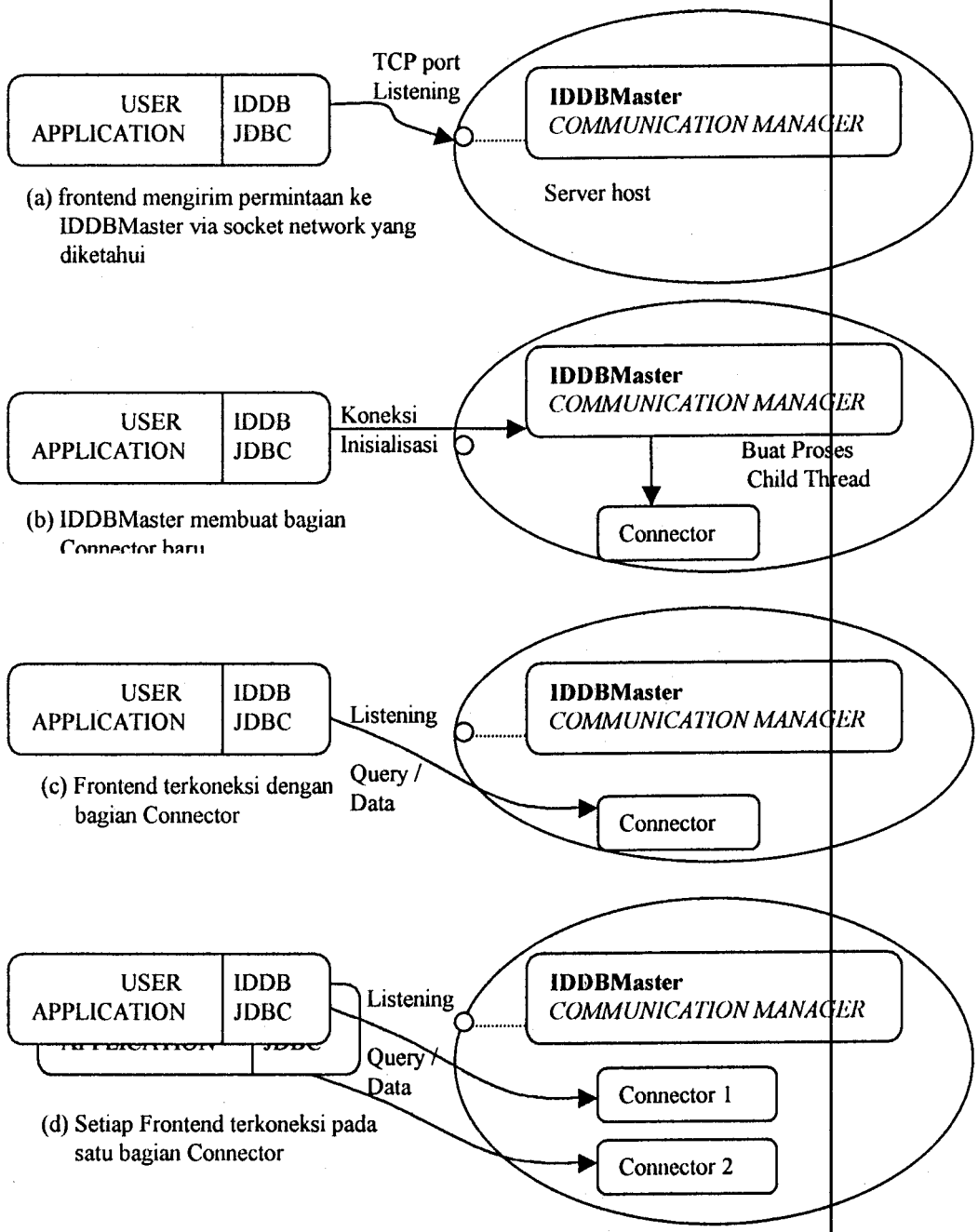
- Sebuah proses *daemon supervisor* (*IDDBMaster*)



Gambar 5.3. Proses pendistribusian query pada IDDB



Gambar 5.4. Arsitektur komunikasi antara client-server-DBMS lokal



Gambar 5.5. Cara koneksi awal antara program aplikasi client dan server IDDBMaster

- Program aplikasi *client* (*frontend*) (misalnya: aplikasi sistim informasi mahasiswa)
- Satu atau lebih bagian connector, dimana masing-masing untuk melayani sebuah koneksi yang masuk.

IDDBMaster bertanggung jawab atas manajemen koleksi basis data yang berada pada sebuah *host*. Program aplikasi *client* (*frontend*) mengakses basis data yang diperlukan dengan membuat sebuah pemanggilan inisialisasi koneksi melalui *driver JDBC IDDB* ke *IDDBMaster*. *Driver JDBC IDDB* tersebut mengirimkan permintaan inisialisasi melalui jaringan komputer ke *IDDBMaster* (Gambar 5.5a), yang akan membuat sebuah class *thread* baru bagian *Connector* (Gambar 5.5b) dan menghubungkan program aplikasi *client* ke sebuah *Connector* baru (Gambar 5.5c). Dari penjelasan tersebut terlihat, antara program aplikasi *client* dengan *server IDDBMaster* dapat saling berkomunikasi tanpa saling tercampur dengan program aplikasi *client* yang lain.

Bagian *Communication Manager* harus selalu jalan untuk menunggu permintaan yang akan masuk. Ia berfungsi sebagai bagian yang mengatur masuk dan keluarnya program aplikasi *client*. Arsitektur ini menjadikan bagian *server IDDBMaster* dapat berada di sebuah komputer yang berbeda dengan program aplikasi *client* yang dapat terletak di komputer-komputer lain dalam jaringan komputer.

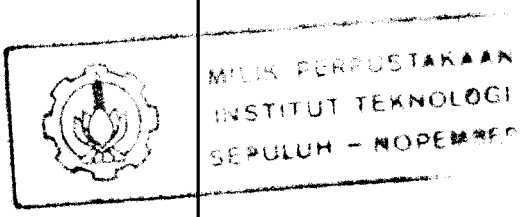
Pengaksesan IDDB secara lokal dapat dilakukan pula, dimana progam aplikasi *client* dapat berhubungan langsung dengan DBMS lokal melalui *driver JDBC IDDB*. *Driver* tersebut dapat memindahkan transfer data ke posisi lokal

dengan pilihan lokal di *file* konfigurasi *driver JDBC IDDB* diberi nilai *on*, sedangkan untuk mengakses basis data secara global melewati *server* variabel lokal di *driver JDBC IDDB* diberi nilai *off* (Gambar 5.2). proses yang dilakukan saat pengaksesan secara lokal sebagai berikut jika diketahui perintah *query* tersebut merupakan *query update*, maka akan dimasukkan catatan perubahan tersebut ke tabel log masing-masing sesuai nama tabel yang dikenai proses *query update* tersebut, lalu meng*update* data nomor urut perubahan tabel tersebut pada tabel *iddbtbcount*. Tabel log tersebut mempunyai nama **log***nama_tabel*.

5.1.3. Protokol Komunikasi

Ketika komunikasi pertama kali dilakukan, *driver JDBC IDDB* mengeluarkan sebuah paket (dinamakan paket inisialisasi) ke *IDDBMaster*. *IDDBMaster* memproses paket inisialisasi dengan menjalankan sebuah *Connector* baru.

Panjang packet : integer [4 byte]	*default : 98*
Kode start up : integer [4 byte]	*default : 9301*
Nama database : string [20 byte]	
Nama pemakai : string [20 byte]	
Password : string [50 byte]	



Setelah paket tersebut diterima lalu dicek oleh *IDDBMaster* apakah valid data-data yang diberikan, kalau valid akan dibalas pesan sukses dengan diberikan karakter 'O', sedangkan kalau tidak valid dibalas dengan pesan error diberikan karakter 'E' ditambah pesan string dengan batas maksimum 1000 byte.

Jika telah diketahui sukses, program aplikasi *client* dapat mulai mengirimkan *query* untuk eksekusi. Jika program aplikasi *client* akan mengakhiri komunikasi

dengan *server*, *driver IDDB JDBC* mengirimkan karakter 'S' ke *server IDDBMaster*, lalu *server* akan menutup saluran koneksi dan menghapus *thread Connector* dari memori.

Abstraksi pengiriman metadata (struktur data) tuple :
kirim karakter 'T'
kirim jumlah kolom tuple dalam integer [4 byte].
Kirim perkolom
Nama Kolom : String [20 byte]
Type : integer [4 byte]
Panjang : integer [4 byte]
Ulangi sampai semua kolom dalam tuple dikirimkan semua.

Abstraksi pengiriman tuple :
Kirim karakter 'D'
Kirim data yang menyatakan bahwa kolom tersebut berisi suatu nilai atau null berupa bit-bit sebanyak jumlah kolom.
Kirim perkolom
Panjang data kolom : integer [4 byte]
Data : byte-byte [sebanyak panjang data kolomg]
Ulangi sampai semua kolom dalam tuple dikirimkan semua.
Transfer data ini diakhiri dengan dikirimkan karakter 'O'.

Saat eksekusi *query*, *driver JDBC IDDB* mengirimkan mengirimkan karakter 'Q' dan string *query* sebanyak maksimum 1000 byte. Lalu pada *server*, jika ia telah selesai melaksanakan *query*, untuk mengirimkan hasil pemprosesan *query*, ia mengirimkan jumlah record yang di*update* atau tuple-tuple dari perintah *query* tipe select. Untuk mengirimkan jumlah tuple dikirimkan karakter 'U', diikuti jumlah tuple dalam integer 4 byte. Untuk mengirimkan hasil perintah *query* tipe select ke *frontend* caranya :

- Kirimkan karakter 'T', diikuti meta data (struktur data) tuple yang akan dikirimkan.
- Kirimkan karakter 'D', diikuti tuple-tuple hasil *query*.

Untuk perintah tambahan yang lain dikirimkan karakter 'C' diikuti string

paling banyak 1000 byte. Balasan dikirim karakter 'D' diikuti jumlah integer, ditambah data sebanyak jumlah tersebut. Sedangkan untuk pesan tambahan dikirimkan karakter 'X' diikuti jumlah integer, ditambah data sebanyak jumlah tersebut.

Dalam segala hal jika terjadi error dikirim karakter 'E', sedangkan untuk warrning dikirim karakter 'W', diikuti string sebanyak maksimum 1000 byte. Sedang setiap kali program sukses diakhiri dengan karakter 'O'. Tipe data string tidak tetap panjangnya, panjang string diketahui pada akhir pengirimaan dikirim karaker berkode ASCII 0.

Untuk mengirimkan tipe auto commit transaksi di kirimkan karakter 'A', lalu diikuti karakter 'Y' jika ya, atau karakter 'N' jika tidak. Sedangkan untuk merubah tipe level transaksi dikirimkan karakter 'L'.

Secara default, seperti pada JDBC spec, auto-commit di set enable, sehingga tidak perlu dilakukan commit secara explisit setelah selesai setiap statement. Commit dan rollback keduanya dapat dilakukan dengan menngunakan pemanggilan metode pada JDBC Connection, atau dengan perintah SQL: commit atau rollback

5.2. Pemodelan dan Perancangan Sistem

Dalam bagian ini dijelaskan mengenai perancangan sistem perangkat lunak. Perancangan tersebut meliputi spesifikasi dan pendekatan yang digunakan dalam sistem perangkat lunak. Pada bagian ini lebih diutamakan penjabaran mengenai IDDBMaster, sedangkan untuk JDBC IDDB, karena dirancang sesuai arsitektur JDBC, dapat di lihat pada document spesifikasi dan arsitektur JDBC yang

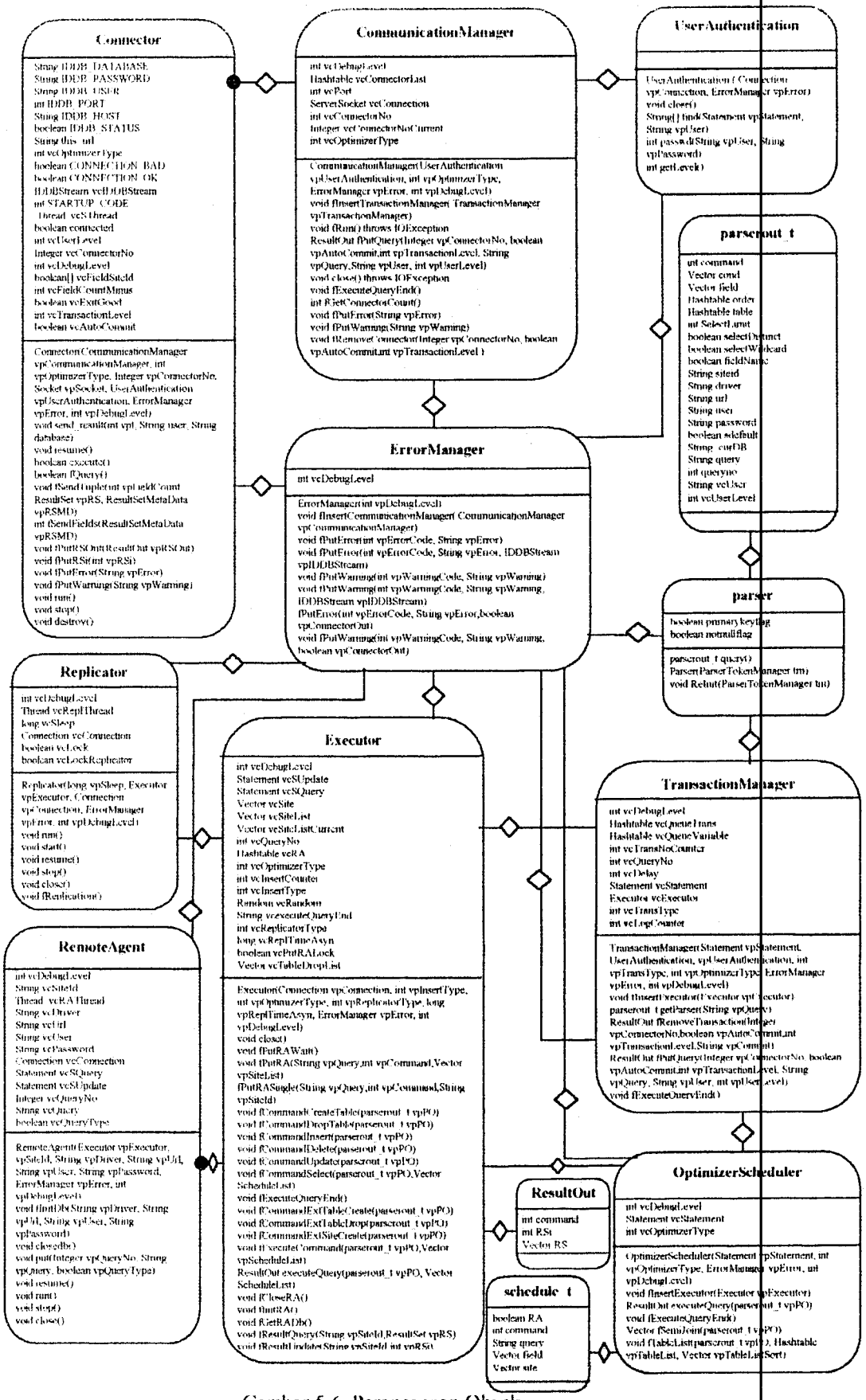
dikeluarkan oleh Sun Microsystem Inc.

5.2.1. Perancangan Obyek

Dalam bagian ini dijelaskan mengenai perancangan obyek dari penjabaran modul-modul yang telah dijelaskan pada bagian sebelumnya. Pendekatan yang digunakan adalah pemprograman berorientasi obyek (Object Oriented Programming). Perancangan dideskripsikan dengan menggunakan Java-*like*.

Obyek-obyek yang terlibat dalam perangkat lunak ini ditunjukkan dalam Gambar 5.6. Obyek-obyek tersebut adalah CommunicationManager, Connector, ErrorManager, Parser, TransactionManager, OptimizerScheduler, Executor, RemoteAgent dan Replicator. Nama dari tiap-tiap obyek tersebut merupakan representasi dari modul-modul yang ada. Berikut ini dijelaskan tentang masing-masing obyek berikut atribut yang dimiliki dan operasi yang ditangani. Implementasi struktur data dari obyek-obyek ini dapat dilihat pada bagian implementasi struktur data.

Obyek Connector jumlahnya dapat lebih dari satu atau tidak ada sama sekali tergantung jumlah koneksi yang diperlukan, ia berhubungan dengan obyek CommunicationManager dan ErrorManager. Obyek ErrorManager merupakan obyek yang berhubungan dengan hampir semua obyek yang ada, gunanya obyek ini untuk menghandle jika kesalahan terjadi dan mengeluarkan pesan yang diperlukan. Obyek CommunicationManger berhubungan dengan UserAuthentication, gunanya untuk melakukan validasi terhadap pemakai yang masuk.



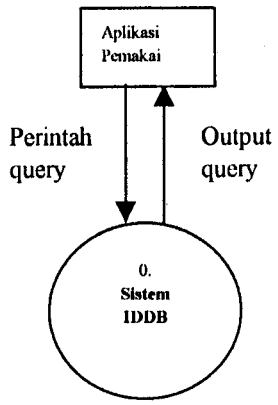
Gambar 5.6. Perancangan Obyek

Obyek TransactionManger berhubungan dengan obyek Parser yang nantinya akan menghasilkan obyek parserout_t yang merupakan struktur data hasil proses *parsing* setiap perintah SQL yang masuk. Obyek Parser ini menggunakan obyek parserout_t untuk menyimpan data-data hasil proses *parsing*. Obyek TransactionManger juga berhubungan dengan obyek OptimizerScheduler. Obyek OptimizerScheduler menghasilkan obyek schedule_t yang merupakan urutan langkah-langkah yang akan dilakukan oleh obyek Executor. Obyek Executor menerima hasil proses dari obyek OptimizerScheduler, dalam prosesnya ia berhubungan dengan obyek Replicator dan obyek RemoteAgent, disamping itu sebagai hasil keluaran proses ia mengeluarkan obyek ResultOut. Obyek RemoteAgent jumlahnya dapat lebih dari satu atau tidak ada sama sekali tergantung jumlah DBMS lokal yang digunakan.

5.2.2. Diagram Aliran Data

Pada DAD level 0 (Gambar 5.6) memperlihatkan pemakai mengakses sistem IDDB melalui suatu aplikasi sistem informasi yang menerima input dari pemakai, ia juga yang akan memberikan output ke pemakai. Sedangkan dari aplikasi sistem informasi memberikan perintah *query* ke sistem IDDB, lalu dari sistem IDDB memberikan hasil *query* ke aplikasi sistem informasi.

Pada DAD level 1 (Gambar 5.8) disini terlihat bahwa sistem IDDB terbagi atas 2 bagian yaitu *driver JDBC IDDB* dengan *IDDBMaster*. Disini *JDBC IDDB* merupakan suatu *interface* yang akan menyampaikan perintah *query* dari aplikasi sistem informasi ke *IDDBMaster* berupa paket-paket data komunikasi awal koneksi,

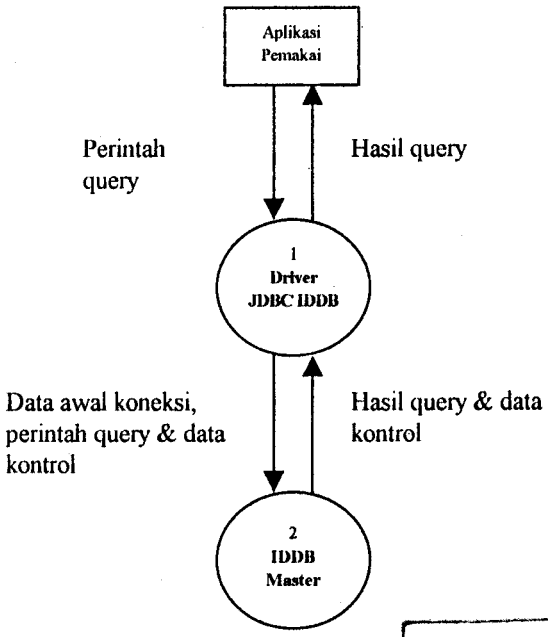


Gambar 5.7. DAD level 0

query dan data kontrol yang dikirimkan ke *IDDBMaster*, sedangkan output dari *IDDBMaster* berupa hasil *query* dan data kontrol yang dikirimkan ke *driver JDBC IDDB* lalu hasil *query* tersebut diberikan ke aplikasi sistem informasi. Data kontrol yang ada digunakan untuk mengatur hubungan antara kedua bagian.

Pada DAD level 2 (Gambar 5.9) ini menjabarkan bagian 1 yaitu bagian *Driver JDBC IDDB*. Bagian ini merupakan implementasi standar arsitektur JDBC yang dimiliki oleh Sun Microsystems, Inc[10] yang telah dijelaskan di bab 4. Ketika program aplikasi *client* meminta koneksi ke IDDB, bagian Driver Manager (komponen JDK1.1.x) mengarahkan permintaan tersebut ke *driver* yang sesuai, jika yang diminta untuk koneksi ke IDDB maka akan diberikan ke *driver* IDDB, dari bagian ini akan meminta bagian koneksi untuk melakukan hubungan pertama kali dengan mengirimkan paket data awal koneksi ke *IDDBMaster*, jika disetujui maka

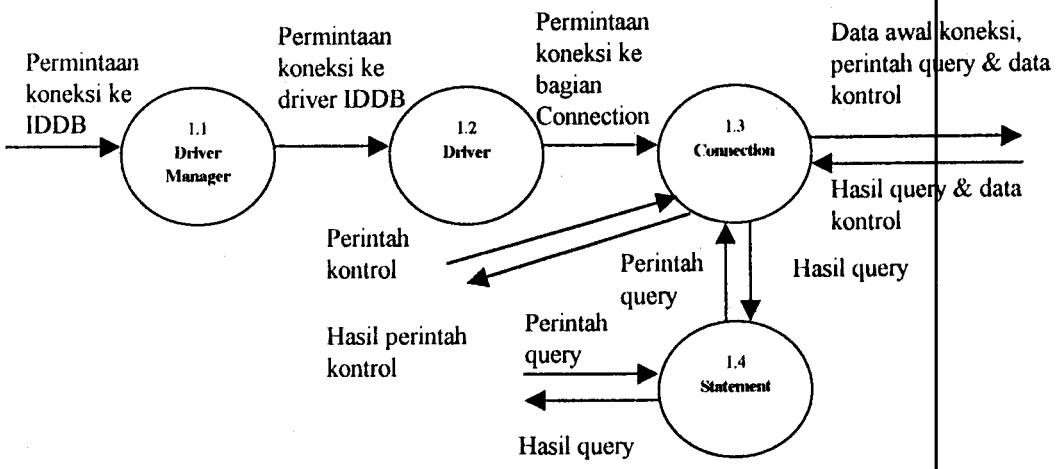
[10] Sun Microsystems, Inc, JDBC: A Java SQL API (1997)



Gambar 5.8. DAD level 1

proses berikutnya siap dilaksanakan, dengan menerima *query* dari bagian statement, yang akan di berikan melalui *Connection* ke *IDDBMaster*, setelah diproses hasil *query* tersebut dikembalikan melalui Connection dan dikembalikan kembali ke bagian *Statement* yang diteruskan ke program aplikasi *client*. Bagian *Connection* juga menerima perintah kontrol yang mengatur koneksi yang jika diperlukan akan meneruskan data tersebut ke *IDDBMaster*, serta *IDDBMaster* akan membalas dengan data kontrol yang sesuai, yang hasilnya dikirimkan balik ke bagian *Connection*.

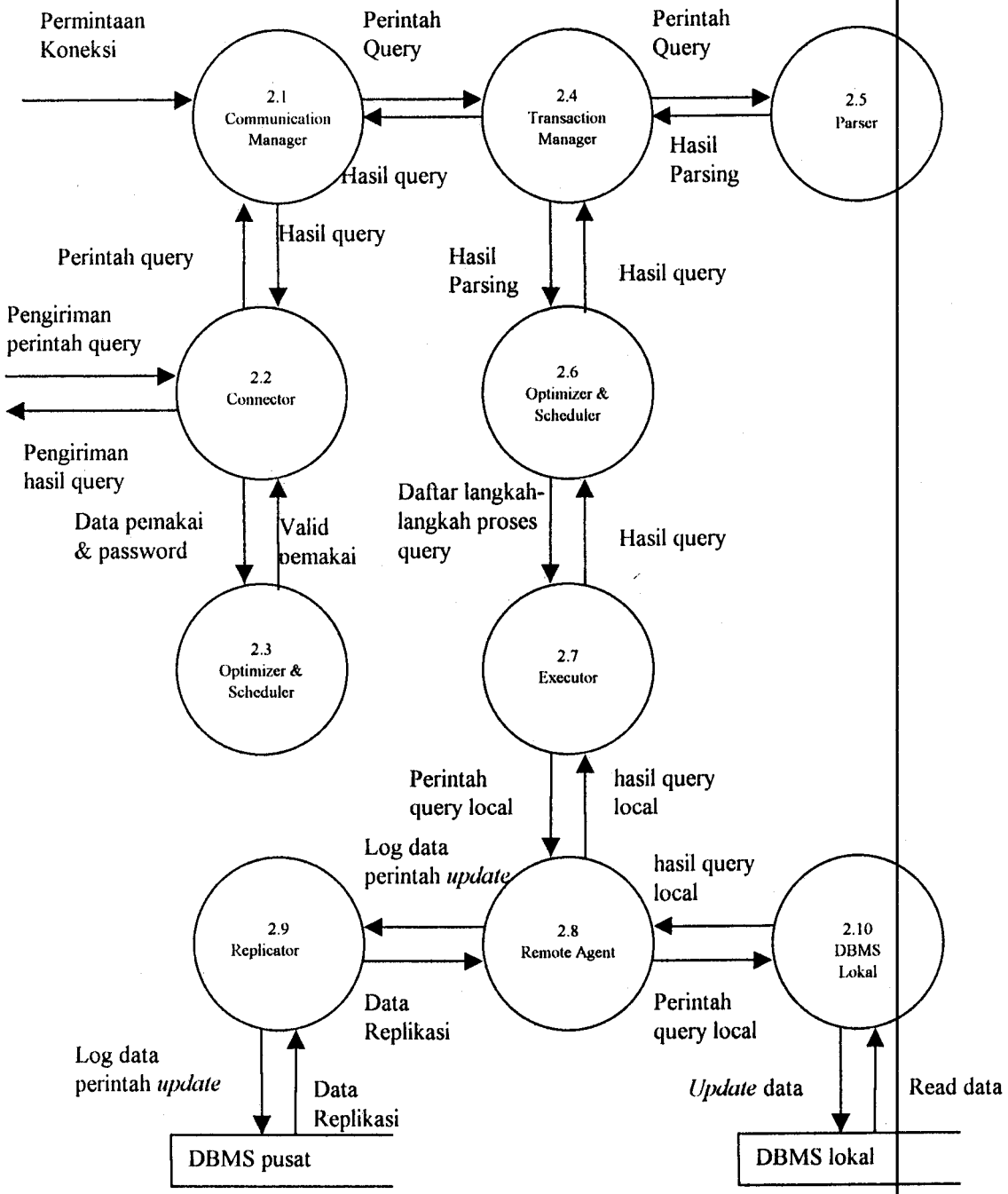
Pada DAD level 2 (Gambar 5.9) ini menjabarkan bagian 2 yaitu bagian *IDDBMaster*. Pada diagram ini terlihat dimana alur data data dari *JDBC IDDB* terpecah jadi dua yaitu permintaan awal koneksi yang masuk ke bagian



Gambar 5.9. DAD level 2, bagian *driver JDBC IDDB*

Communication Manager, sedang permintaan dan pengiriman *query* masuk ke bagian *Connector*. Saat awal koneksi dibentuk suatu bagian *Connector* yang baru dengan melakukan validasi pemakai dan *password*, jika valid maka *Connector* dapat melanjutkan proses penerimaan *query*.

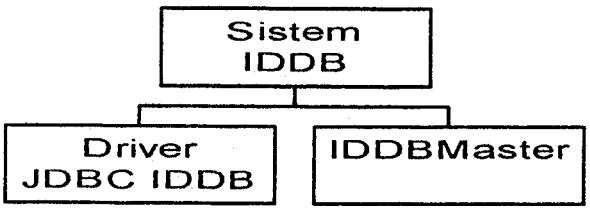
Perintah *query* yang diterima oleh *Connector* diteruskan ke *Communication Manager* yang akan diteruskan ke *Transaction Manager*, dari sini akan diberikan ke *Parser*, sedang hasil parsing ini dikirim kembali ke *Transaction Manager*, hasil parsing tersebut dikirimkan ke bagian *Optimizer&Scheduler* untuk menghasilkan suatu langkah-langkah urutan proses *query* yang akan diberikan ke bagian Executor, disini *query* akan dijalankan ke *Remote Agent* dan jika *Replikator* dinyalakan, maka data akan disimpan di bagian Replikasi. Sedangkan untuk hasil output berupa hasil *query* akan dikirimkan melewati bagian-bagian yang diatasnya sampai ke *Transaction Manager* kembali diberikan ke *Communication Manager* yang akan mengirimkan ke *Connector* yang sesuai yang nantinya akan dikirimkan berupa paket-paket data yang dikenali oleh *driver JDBC IDDB*.



Gambar 5.10 DAD level 2, bagian IDDBMaster

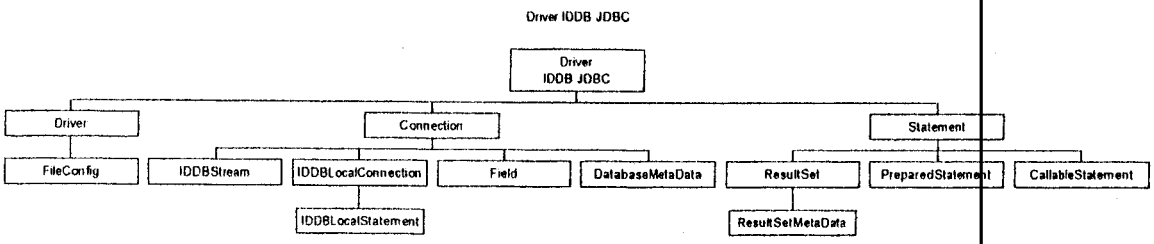
5.2.3. Hirarki Diagram

Pada hirarki diagram level 1 (Gambar 5.10) disini dibagi atas dua bagian yaitu *Driver JDBC IDDB* dan *IDDBMaster*.



Gambar 5.11. Hirarki diagram level 1

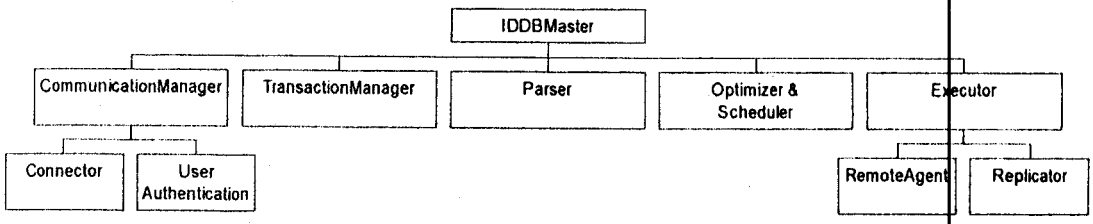
Sedangkan pada hirarki diagram level 2 (Gambar 5.11) merupakan diagram untuk bagian *Driver JDBC IDDB* yang didasarkan pada arsitektur JDBC yang dikeluarkan oleh Sun MicroSystems, Inc. Disini terdapat beberapa bagian tambahan, seperti FileConfig gunanya untuk mengambil data-data konfigurasi dari file JDBC IDDB.ini sebagai data konfigurasi untuk *driver* ini.



Gambar 5.12. Hirarki diagram level 2, bagian driver JDBC IDDB

IDDBStream gunanya untuk sebagai bagian yang mengolah proses komunikasi lewat jaringan komputer antara JDBC IDDB dengan *server* IDDBMaster. IDDBLocalConnection gunanya untuk menghubungkan bagian Connection ke bagian Connection dari *driver* JDBC DBMS lokal. IDDBLocalStatement gunanya untuk mengatur proses *update* data secara lokal untuk sinkronisasi replikasi data.

Sedangkan untuk hirarki diagram level 2 bagian *IDDBMaster* dapat dilihat di Gambar 5.12



Gambar 5.13 Hirarki diagram level 2, bagian IDDBMaster

5.3. Implementasi Sistem

Aspek pemrograman yang dibuat dalam prototipe sistem basis data terdistribusi ini meliputi implementasi struktur program yang telah dirancang ke dalam kode-kode bahasa pemrogram yang dipakai. Berikut akan dijelaskan tentang bahasa pemrograman yang digunakan, dan struktur data peprograman *server* (sedangkan struktur data basis data utama yang digunakan dapat dilihat pada lampiran B).

5.3.1. Kebutuhan Sistem

Prototipe sistem basis data terdistribusi ini dibuat dengan beberapa batasan sehubungan dengan perangkat keras (hardware) dan perangkat lunak (software) yang dibutuhkan agar sistem dapat bekerja dengan baik. Kebutuhan-kebutuhan tersebut dijelaskan sebagai berikut :

5.3.1.1. Kebutuhan Perangkat Keras

Perangkat keras yang digunakan ialah komputer sebagai primary *site* dan komputer-komputer untuk DBMS lokal dan untuk menjalankan aplikasi *client*. Disini

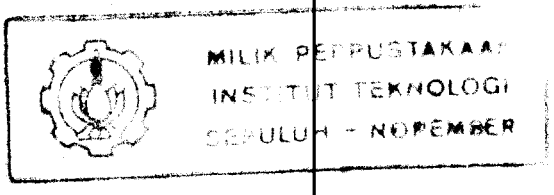
diperlukan juga jaringan komputer yang selalu terjaga dari kemungkinan kegagalan komunikasi jaringan komputer, untuk mengirimkan data-data yang ada.

5.3.1.2. Kebutuhan Perangkat Lunak

Sistem ini dikembangkan dengan bahasa Java sehingga tidak tergantung pada suatu jenis platform tertentu, sehingga dapat berbeda tipe prosessor perangkat keras dan operating sistem, asal telah diinstall JVM (Java Virtual Machine) menurut platform masing-masing, disini dapat digunakan Java Development Kit (JDK) yang dapat digunakan compiler dan debugging atau Java Runtime Environment (JRE), yang hanya digunakan untuk menjalankan program saja, atau program lain yang dibuat perusahaan diluar Sun Microsystems, Inc.

Pada waktu development dan testing aplikasi dikerjakan dengan Operating Sistem Microsoft Windows 95. Program sistem IDDB ini dibuat dengan menggunakan Symantec Café 1.8. Bagian parser dibuat dengan menggunakan perangkat lunak Java Compiler-Compiler (JavaCC), disini dimasukkan input grammar SQL minimum kemudian oleh JavaCC di-generate modul parser dalam bahasa Java.

Program DBMS dibutuhkan untuk digunakan sebagai primary *site* dan lokal, tidak harus spesifik DBMS tertentu asalkan telah mempunyai *driver* JDBC. Misalnya Interbase, Oracle, Informix, Sybase, Microsoft SQL *Server*, Postgresql, MySQL, via ODBC (misal: Access, dBase, FoxPro dengan menggunakan JDBC-ODBC bridge).



5.3.2. Struktur Data

Pada bagian ini lebih diutamakan penjabaran struktur data mengenai IDDBMaster, sedangkan untuk JDBC IDDB, karena dirancang sesuai arsitektur JDBC, dapat di lihat pada document spesifikasi dan arsitektur JDBC yang dikeluarkan oleh Sun Microsystem Inc. Sesuai dengan rancangan program yang telah dijelaskan, maka struktur data dari program sistem ini dibuat sebagai berikut:

```
class parserout_t
{  int command;
   Vector cond;
   Vector field ;
   Hashtable order;
   Hashtable table;

   String siteid, driver, url, user, password;
   boolean sdefault;
   String  curDB;
   String query;
   int queryno;

   String vcUser;
   int vcUserLevel;
}
```

Struktur "parserout_t" dipergunakan sebagai struktur data-data keluaran hasil parsing dari modul parser. Data parsing dibagi menjadi empat bagian utama yaitu : "table" untuk data tabel yang diproses saat *query*, ia menggunakan struktur Hashtable untuk mempermudah pencarian yang didalamnya digunakan struktur data "tname_t". "field" untuk data-data yang berhubungan dengan *field* dari suatu tabel, ia menggunakan struktur Vector untuk penyimpanan *field-field* lebih dari satu, yang di dalamnya digunakan struktur data "field_t". "cond" berisi kondisi / aturan pencarian data, ia menggunakan struktur Vector, yang didalamnya digunakan struktur data "cond_t". "order" berisi data untuk proses sorting, ia menggunakan struktur Hashtable yang didalamnya digunakan struktur data "order_t". "command" untuk menyatakan tipe perintah. Sedangkan variabel yang lain merupakan variabel

tambahan untuk proses *query* yang belum dimasukkan.

```
class tname_t
{ String name, cname;
}
```

Struktur "tname_t" digunakan untuk menyimpan nama tabel dan aliasnya

```
class field_t
{ String table, name;
  val_t value;
  int type, length;
  cond_t cond;
  boolean notnullflag;
  boolean primarykeyflag;
}
```

Struktur "field_t" digunakan untuk menyimpan struktur data *field-field* tabel

```
class cond_t
{ String table, name;
  val_t value;
  int op, bool, type, length;
  boolean hasbeencreatetable;
}
```

Struktur "cond_t" digunakan untuk menyimpan aturan kondisi table.

```
class ident_t
{ String seg1, seg2;
};
```

Struktur "ident_t" digunakan untuk menyimpan variabel yang digunakan pada perintah *query*.

```
class val_t
{ intVal;
  String  charVal;
  float   floatVal;
  ident_t identVal;
  int     type, datalen;
  boolean nullVal;
};
```

Struktur "val_t" digunakan untuk menyimpan nilai-nilai yang ada pada perintah *query*.

```
class schedule_t
{ boolean RA;
  int command;
  String query;
  Hashtable field;
  Vector site;
}
```

Struktur "schedule_t" merupakan sturktur data yang menyimpan data-data yang digunakan sebagai hasil output dari optimiser yang digunakan oleh operasi select dengan algoritma semi-join. Struktur data ini nantinya yang akan dijalankan oleh eksekutor di fungsi fCommandSelect. Variabel "RA" menyatakan apakah proses dilakukan oleh bagian Remote Agent atau di pusat. "command" menyatakan tipe command. "query" menyatakan perintah *query*. "field" menyatakan *field-field* yang terlibat. "site" menyatakan *site-site* mana saja yang terlibat.

```
class ResultOut
{ int command
  int RSi
  Vector RS
}
```

Struktur "ResultOut" digunakan untuk menyimpan nilai-nilai hasil keluaran dari obyek Executor hasil proses sebuah perintah *query*.

Pada bagian berikut dijelaskan mengenai kegunaan fungsi-fungsi yang ada pada tiap-tiap class.

```
class CommunicationManager
{  CommunicationManager(UserAuthentication vpUserAuthentication, int vpOptimizerType, ErrorManager
        vpError, int vpDebugLevel)
   void fInsertTransactionManager( TransactionManager vpTransactionManager)
   void fRun()
   ResultOut fPutQuery(Integer vpConnectorNo, boolean vpAutoCommit,int vpTransactionLevel, String
        vpQuery,String vpUser, int vpUserLevel)
   void close() throws IOException
   void fExecuteQueryEnd()
   int fGetConnectorCount()
   void fPutError(String vpError)
   void fPutWarning(String vpWarning)
   void fRemoveConnector(Integer vpConnectorNo, boolean vpAutoCommit,int vpTransactionLevel )
}
```

Class "CommunicationManager" mempunyai fungsi-fungsi : CommunicationManger untuk inisialisasi class saat pembentukan class pertama kali. fInsertTransactionManger digunakan untuk memasukkan obyek TransactionManger. fRun untuk menjalankan obyek ini. fPutQuery untuk memasukkan sebuah *query* dari obyek Connector. close untuk menutup obyek. fExecuteQueryEnd untuk mengakhiri

query. fGetConnectorCount untuk menghitung jumlah obyek Connector yang aktif. fPutError dan fPutWarning untuk menampilkan kesalahan dan peringatan ke obyek Connector. fRemoveConnector untuk menghapus sebuah obyek Connector.

```
class Connector
{  Connector(CommunicationManager vpCommunicationManager, Integer vpno,Socket vpSvrSocket,
       UserAuthentication vpUserAuthentication, IDDBError vpError)
   void send_result(int i)
   boolean execute()
   void SendTuple(int nf, ResultSet rs, ResultSetMetaData rsmd)
   int SendFields(ResultSetMetaData rsmd)
   void putrs(ResultSet rs)
   void putError(String err)
}
```

Class "Connector" mempunyai fungsi-fungsi : Connector untuk inisialisasi class saat pembentukan class pertama kali. send_result untuk mengirimkan result ke *client*. SendTuple untuk mengirim struktur tuple-tuple ke *client*. Send Field untuk mengirim satu *field* ke *client*. PutError untuk mengirimkan pesan error ke *client*.

```
class UserAuthentication
{ UserAuthentication(Statement st)
  String[] find(Statement s,String user)
  int passwd(String user, String password)
}
```

Class "UserAuthentication" mempunyai fungsi-fungsi : UserAuthentication sebagai constructor class. find untuk menemukan record user dalam tabel iddbuser. passwd untuk melakukan pengecekan user dan *password* yang diberikan saat awal koneksi.

```
class TransactionManager
{ TransactionManager(Statement vpStatement, IDDBError vpEr, int vpDebug)
  parserout_t fGetParser(String vpQuery)
  ResultOut fPutQuery(Integer vpConnectorNo, String vpQuery, String vpUser, int vpUserLevel)
}
```

Class "TransactionManager" mempunyai fungsi-fungsi : TransactionManager sebagai constructor class. fGetParser untuk melakukan parsing pada perintah *query* yang diberikan. fPutQuery untuk mengelompokkan perintah *query* menurut nomor transaksi yang dipunyainya, fungsi ini sebagai *interface* dari modul lain.

```
class Parser
{ Parser(ParserTokenManager tm)
  void ReInit(ParserTokenManager tm)
  parserout_t query()
}
```

Class "Parser" mempunyai fungsi-fungsi : Parser sebagai constructor class. ReInit untuk melakukan inisialisasi data input class Parser. query untuk melakukan proses *parsing*.

```
class OptimizerScheduler
{ OptimizerScheduler(java.sql.Statement stm, IDDBError vpEr, int vpIDebug)
  ResultOut fExecuteQuery(parserout_t po)
  void fProcessSemiJoin()
  void fProcessTableList(Hashtable tablelist, Vector tablelistsort)
}
```

Class "OptimizerScheduler" mempunyai fungsi-fungsi : OptimizerScheduler sebagai construktor class. fExecuteQuery untuk menjalankan *query* dengan perintah bertipe select, fungsi ini sebagai *interface* dari modul lain. fProcessTableList untuk membentuk daftar tabel-tabel yang digunakan pada fungsi processSemiJoin. fProcessSemiJoin untuk membentuk suatu schedule operasi select menurut algoritma optimasi semi-join.

```
class Executor
{ Executor(Connection vpDb, IDDBError vpEr, int vpIDebug)
  void close()
  void fPutRA(String vpQuery,int vpCommand,Vector vpSite)
  int fSingleExecuteUpdate(String vpSiteId,String vpQuery)
  void fCommandCreateTable() throws SQLException
  void fCommandDropTable()throws SQLException
  void fCommandInsert()
  void fCommandDelete()
  void fCommandUpdate()
  void fCommandSelect(Vector ScheduleList)
  void fCommandExtTableCreate()
  void fCommandExtTableDrop()
  void fCommandExtSiteCreate()
  void fCommandExtSiteDrop()
  void fExecuteCommand(Vector vpScheduleList)
  ResultOut fExecuteQuery(parserout_t vpPO, Vector vpScheduleList)
  void fCloseRA()
  void fInitRA()
}
```


Class "Executor" mempunyai fungsi-fungsi : Executor sebagai construktor class. Close untuk menutup mengakhiri proses dalam class. fPutRA untuk

mengeksekusi *query* di Remote Agent. fSingleExecuteUpdate untuk mengeksekusi *query* tunggal di Remote Agent. fcommandCreateTable untuk membuat sebuah tabel. fcommandDropTable untuk menghapus sebuah tabel. fCommandInsert untuk memasukkan sebuah record. fCommandDelete untuk menghapus record. fCommandUpdate untuk mengubah record. fCommandSelect untuk melakukan perintah select. fCommandExtTableCreate untuk melakukan perintah extension table create. fCommandExtTableDrop untuk melakukan perintah extension table drop. fCommandExtSiteCreate untuk melakukan perintah extension site create. fCommandExtSiteDrop untuk melakukan perintah extension site drop. fExecuteCommand untuk mengeksekusi *query* menurut tipe command. fExecuteQuery untuk menjalankan *query* dengan perintah fungsi ini sebagai *interface* dari modul lain. fCloseRA untuk menutup semua Remote Agent. fInitRA untuk menginisialisasi semua Remote Agent,

```
class RemoteAgent
{  RemoteAgent(IDDBError vpError, String vpSiteId, String vpDriver, String vpUrl, String vpUser, String
       vpPassword)
   void close()
   void fInitDb(String vpDriver, String vpUrl, String vpUser, String vpPassword)
   void fCloseDb()
   void fPutQuery(int vpQueryNo, String vpQuery)
}
```

Class "RemoteAgent" mempunyai fungsi-fungsi : RemoteAgent sebagai constructor class. fPutQuery untuk memasukkan *query* ke Remote Agent, fungsi ini sebagai *interface* dari modul lain. Close untuk menutup mengakhiri proses dalam class. fInitDB untuk melakukan inisialisasi DBMS lokal. fCloseDB untuk mengakhiri / menutup DBMS lokal.

```
class Replicator
{  Replicator(long vpSleep, Executor vpExecutor, Connection vpConnection, ErrorManager vpError, int
       vpDebugLevel)
   void fReplication()
}
```

Class "Replicator" mempunyai fungsi-fungsi : Replicator sebagai constructor class. fReplication untuk melakukan proses replikasi.

```
class ErrorManager
{ ErrorManager(int vpDebugLevel)
  void fPutError(int vpErrorCode, String vpError)
  void fPutError(int vpErrorCode, String vpError, IDDBStream vpIDDBStream)
  void fPutWarning(int vpWarningCode, String vpWarning)
  void fPutWarning(int vpWarningCode, String vpWarning, IDDBStream vpIDDBStream)
  void fPutError(int vpErrorCode, String vpError,boolean vpConnectorOut)
  void fPutWarning(int vpWarningCode, String vpWarning, boolean vpConnectorOut)
}
```

Class "ErrorManager" mempunyai fungsi-fungsi : ErrorManager sebagai constructor class. fReplication untuk melakukan proses replikasi. fPutError untuk menampilkan kesalahan menurut tipe data masing-masing. fPutWarning untuk menampilkan pesan menurut tipe data masing-masing.

5.3.3. Implementasi Algoritma

Disini dibahas mengenai implementasi dalam pseudocode dalam bentuk Java-*like*, yang dibahas hanya beberapa prosedure-prosedure yang penting. Pertama kali dibahas mengenai prosedure fPutQuery untuk pengatur transaksi yang ada dalam class Transaction Manager. Disini terlihat jika perintah bertipe commit akan dilakukan proses commit dengan melakukan pengecekkan pada nilai timestamp tiap-tiap variabel (disini mewakili tabel), jika bertipe rollback akan dihapus semua variabel-variabel yang pernah dimasukkan, sedangkan jika bukan perintah tersebut yaitu perintah 'create, drop, atau extended' akan langsung dilakukan, sedangkan jika perintah bertipe *update* (insert, update, delete) akan dimasukkan ke dalam queue *update* tiap-tiap tabel, sedangkan jika bertipe read maka data akan langsung dibaca.

```
public ResultOut fPutQuery( Integer vpConnectorNo, boolean vpAutoCommit,int vpTransactionLevel,
                String vpQuery, String vpUser, int vpUserLevel)
```

```
{  vcQueryNo++;
   parserout_t vfPO = getParser(vpQuery); // parsing query
   if(transaksi bertipe single AND sudah ada transaksi yang masuk)
   {  Tampil Error
      return vfRSOut;
   }
   if((query tidak tipe create,drop,atau extended) AND tidak AutoCommit AND (TransType tidak TRANS_NONE))
   {  cari trans sesuai vpConnectorNo, kalau tidak ada buat baru
      if(vfPO.command==COMMAND_COMMIT)
      {  for (ambil semua perintah query pada tranksaksi)
         {  for(ambil semua nama tabel yang ada pada perintah query)
            {  ambil variabel dari nama tabel
               if(vfTrans.commit)
               {  if(vfCommand.po.command==COMMAND_SELECT)
                  {  tunggu jika masih ada transaksi yang mengupdate variabel tetapi belum commit
                     if(variabel telah dibaca sebelum diupdate transaksi sebelumnya)
                        vfTrans.commit=false;
                     else if(ada update dari trans sesudahnya)
                        set ts(read x) = max (ts(read x), ts(Ti))
                  }
                  else
                  if(vfCommand.po.command==COMMAND_UPDATE)
                  {  tunggu jika masih ada transaksi yang mengupdate variabel tetapi belum commit
                     if(ada update dari trans sesudahnya)
                        vfTrans.commit=false;
                     else if(telah pernah dibaca oleh transaksi sesudahnya)
                        vfTrans.commit=false;
                     else
                     {
                        set ts(update x)=ts(T)
                        operasi UPDATE
                     }
                  }
               }
               hapus semua variable transaksi di bagian update variable
            }
            hapus transaksi, di queueTrans
         }
      } else if(vfPO.command==COMMAND_ROLLBACK)
      {
         hapus elemen semua variable queue UPDATE.
         hapus elemen vfTrans, di queueTrans
      } else {
         masukkan command ke vfTrans
         for(ambil semua nama tabel yang ada pada perintah query)
         {
            ambil variabel dari nama tabel
            if(vfPO.command==COMMAND_SELECT)
            {  if(ada update dari trans sesudahnya)
                  vfTrans.commit=false;
               else
                  OPERASI READ
            }
            else if(vfPO.command>COMMAND_SELECT)
            {  if(ada update dari trans sesudahnya)
                  vfTrans.commit=false;
               else if(pernah dibaca trans sesudahnya)
                  vfTrans.commit=false;
               else
                  masukkan command update ke queue update variabel
            }
         }
      }
   }
} else {
```

```
    if(perintah bertipe select, insert, update, atau delete)
    {  for(ambil semua nama tabel yang ada pada perintah query)
       {  ambil variabel dari nama tabel
          if(vfPO.command==COMMAND_SELECT)
             set ts(read x) = max (ts(read x), ts(Ti))
          else
             set ts(update x)=ts(T)
       }
    }
    Lakukan query vpQuery
  }
  return vfRSOut;
}
```

Procedure selanjutnya membahas mengenai pseudocode dari bagian replikasi. Procedure fReplication disini membandingkan apakah jumlah counter log tabel di bagian replikasi sama dengan di data utama di DBMS lokal, jika tidak, berarti ada perubahan data di DBMS lokal, maka lakukan *update* data replikasi.

```
public void fReplication()
{
  ResultSet vfRS = ambil data tbname, siteid dari tabel iddbtbsite
  while(ada data di vfRS)
  {
    ResultSet vfRS2 = ambil data icounter dari tabel iddbtbcount dimana isiteid=siteid and itbname=tbname
    ResultSet vfRSRA = ambil data icounterRA dari tabel iddbtbcount di DBMS lokal dimana isiteid=siteid and
      itbname=tbname
    if(icounter<icounterRA)
    {  vfRSRA = ambil data log dari tabel log'tbname' dimana icounterRA>=icounter dan isiteid<>'*'
      while( ada data di vfRSRA)
        update data replikasi dengan data dari tabel log pada DBMS lokal
      update tabel iddbtbcount dengan nilai icounter=icounterRA dimana isiteid=siteid and itbname=tbname
    }
  }
}
```

Procedure berikutnya membahas mengenai pseudocode dari bagian optimasi yang menggunakan algoritma semijon. Procedure fSemijoin ini akan menghasilkan daftar langkah-langkah *query* yang akan dilakukan oleh bagian Executor. Pertama-tama dibuat daftar yang diperlukan kemudian proses semua *field* dan aturan join yang digunakan, lalu buat perintah *query* yang dimasukkan dalam ScheduleList.

```
public Vector fSemiJoin(parserout_t vpPO)
{
   Vector vfScheduleList;
   bentuk daftar tabel vfQueueTabel, daftar field vfQueueField, daftar join vfQueueJoin
   for(t=0; t < jumlah tabel dalam daftar tabel - 1)
   {
      vfTName = vfQueueTabel
      proses field-field yang digunakan
```

```
        proses aturan join
        masukkan query 'create tabel tmp'+vfTName di vfScheduleList
        masukkan query untuk RA'create tabel tmp'+vfTName di vfScheduleList
        masukkan query untuk RA'select tabel '+vfTName di vfScheduleList
        masukkan query 'insert tabel tmp'+vfTName di vfScheduleList
        masukkan query untuk RA'insert tabel tmp'+vfTName di vfScheduleList
    }
    masukkan query untuk'select tabel '+vfTName di vfScheduleList
    masukkan query untuk melakukan drop semua tabel tmp di vfScheduleList
}
```

BAB VI

UJI COBA DAN EVALUASI SISTEM

Pada bab ini akan dibahas mengenai uji coba sistem dan analisanya. Sebelumnya dijelaskan faktor-faktor yang berpengaruh terhadap proses uji coba. Kemudian dibahas mengenai hasil uji coba serta analisanya. Selain itu juga sebuah program aplikasi contoh, yang menggunakan fasilitas basis data terdistribusi dari sistem IDDB.

6.1. Uji Coba

Tujuan pengujian sistem adalah untuk mengetahui validitas, efektifitas dan unjuk kerja sistem yang dibuat. Dengan uji coba ini dapat disimpulkan sejauh mana efektifitas metode yang ditawarkan mampu menyelesaikan permasalahan yang ada.

6.1.1. Hal-hal yang mempengaruhi uji coba

Faktor-faktor yang mempengaruhi kinerja sistem IDDB ini terdapat dua hal. Faktor yang pertama ialah pengaruh di luar sistem IDDB, yaitu lingkungan di mana sistem IDDB dijalankan, hal yang berpengaruh disini ialah kinerja komputer yaitu kecepatan prosesor, *hard disk*, memori, serta kecepatan jaringan komputer

dalam mendistribusikan data juga mempengaruhinya. Sistem operasi yang digunakan berpengaruh pula, sebagai contoh Unix yang umumnya berbasis teks, pada umumnya kinerjanya lebih cepat dari pada Windows yang berbasis grafik, karena Windows dalam prosesnya juga harus memproses tampilan dalam mode grafik yang akan menambah beban komputer. Sedangkan program yang jalan bersamaan dengan sistem IDDB ini tentunya akan mempengaruhi hasil pengujian, hal ini karena sistem operasi yang ada sifatnya *multi tasking* yang akan memproses secara bersamaan semua program yang ada, dengan cara membagi pekerjaan ke prosesor secara bergantian.

Sedangkan faktor dari dalam sistem IDDB sendiri ialah penentuan konfigurasi pada IDDB. Hal tersebut seperti perubahan parameter konfigurasi pada bagian mode optimasi *query*, mode transaksi, mode *insert*, dan mode replikasi.

6.1.2. Tahap pengujian

Pada bagian ini dijelaskan mengenai konfigurasi perangkat keras dan perangkat lunak yang mempengaruhi sistem saat uji coba.

6.1.2.1. Konfigurasi perangkat keras

Perangkat keras yang digunakan *primary site* ialah komputer PC IBM *compatible*, dengan prossesor Pentium 133 Mhz dan memori 16 Mega byte, sedangkan komputer yang digunakan untuk DBMS lokal dan program aplikasi *client* ialah komputer PC IBM *compatible*, prossesor Pentium 100 Mhz, dengan

memori 16 Mega byte. Jaringan komputer yang ada menggunakan *ethernet* biasa dengan kecepatan maksimal 10 Mbit / sec menggunakan kabel *coaxial* sebagai media pengiriman.

6.1.2.2. Konfigurasi perangkat lunak

Sistem operasi yang digunakan ialah Windows 95 dan Linux paket Slackware 3.3. Untuk menjalankan program Java di Windows 95 digunakan Java Development Kit 1.1.3 (JDK), sedangkan di Linux digunakan JDK 1.1.1.

DBMS yang digunakan sebagai DBMS lokal pada sistem IDDB ialah Interbase 4.2 ditambah dengan *driver* JDBC-nya dengan InterClient 1.12, PostgreSQL 6.3.2 dengan *driver* JDBC, serta MySQL 3.21.20 dengan *driver* JDBC, serta Access (dengan MSJET35.dll) menggunakan *driver* JDBC-ODBC bridge. Interbase dan Access merupakan DBMS yang jalan di Windows 95 sedangkan PostgreSQL dan MySQL merupakan DBMS yang jalan di Linux.

6.1.3. Hasil Pengujian

Cara melalukan pengujian pada sistem ini ialah dengan membuat program dalam bahasa Java yang mengakses basis data dengan menggunakan driver JDBC masing-masing. Uji coba pertama kali dibandingkan kecepatan DBMS yang ada yang digunakan sebagai DBMS lokal (Tabel 6.1). Disini dibuat 4 buah tabel, lalu tiap-tiap operasi bertipe *update* (insert, update, delete) dilakukan sebanyak 30 kali, sedangkan operasi bertipe *read* (select) dibedakan menjadi 4 macam, dimana mengakses 1 tabel, 2 tabel, 3 tabel dan 4 tabel, disini aturan join digunakan untuk

mengambil data dari tabel-tabel yang diakses secara bersamaan. Pengukuran dilakukan dalam satuan milisecond.

Tabel 6.1. Perbandingan kecepatan DBMS yang digunakan.

No	Perintah	Jumlah	Access		Interbase		PostgreSQL		MySQL	
			Total	Rata-	Total	rata-rata	total	rata-rata	Total	rata-rata
1	Insert	30	920	30	11960	398	1590	53	390	13
2	Update	30	1110	37	11530	384	2090	69	390	13
3	Delete	30	720	24	13590	453	2040	68	60	2
4	Select 1 tabel	4	1540	385	2410	602	1530	382	210	42
5	Select 2 tabel	2	380	190	830	415	1870	935	170	85
6	Select 3 tabel	1	160	160	550	550	820	820	330	330
7	Select 4 tabel	1	380	380	550	550	1380	1380	440	440
8	Create table	4	600	150	5760	1440	1970	492	110	27
9	Drop table	4	220	55	2860	715	440	110	110	27
	Total			1411		5507		4309		979

Pada tabel-tabel berikut (tabel 6.2.-6.7.) pengujian dilakukan pada sistem IDDB. Sebagai konfigurasi standar sistem IDDB saat melakukan perbandingan-perbandingan tersebut ialah optimasi mode semijoin, replikasi mode synchronous, insert mode fragment, dan transaksi mode concurrent. Konfigurasi tersebut pada saat pembandingan dilakukan perubahan pada bagian konfigurasi yang dilakukan pembandingan saja.

Pada Tabel 6.2. dibandingkan kecepatan pengaksesan data pada sistem IDDB, jika digunakan perbedaan jumlah DBMS lokal.Disini Access digunakan sebagai DBMS dan DBMS pusat. Perbandingan dilakukan melalui penggunaan 1 DBMS, 2 DBMS, dan 3 DBMS. Pada pengujian sistem IDDB ini terdapat empat perintah tambahan yang diuji yaitu 'extended table create', 'extended table drop', 'extended site create', dan 'extended site drop'.

Tabel 6.2. Perbandingan kecepatan berdasarkan jumlah DBMS lokal.

No	Perintah	Jumlah	Access 1		Access 2		Access 3	
			Total	rata-rata	total	rata-rata	total	rata-rata
1	Insert	30	33120	1104	52880	1762	78450	2615
2	Update	30	24440	814	30740	1024	39280	1309
3	Delete	30	29370	979	36530	1217	50080	1669
4	Select 1 tabel	4	9340	2335	16150	4037	30050	7512
5	Select 2 tabel	2	9010	4505	14390	7195	23390	11695
6	Select 3 tabel	1	5320	5320	8950	8950	13080	13080
7	Select 4 tabel	1	8620	8620	15320	15320	19660	19660
8	Create table	4	3520	800	4170	1042	5060	1265
9	Drop table	4	2540	635	3300	825	4560	1140
10	Extended table create		4 3780	945	8 10220	1277	12 17910	1492
11	Extended table drop		4 3520	880	8 8190	1023	12 18350	1529
12	Extended site create		1 6870	6870	2 7850	3925	3 8840	2946
13	Extended site drop		1 4940	4940	2 5990	2995	3 7300	2433
	Total			38747		50592		68345

Tabel 6.3 Perbandingan penggunaan multi DBMS.

No	Perintah	Jumlah	Access 1		Access 2		PostgreSQL	
			Total	rata-rata	total	rata-rata	Total	rata-rata
1	Insert	30	150500	5012	64580	2152	188450	6281
2	Update	30	53650	1788	62140	2071	70630	2354
3	Delete	30	61510	2050	61510	2050	56580	5050
4	Select 1 tabel	4	26210	6552	20600	5150	39110	9777
5	Select 2 tabel	2	50040	25020	44370	22185	59260	29630
6	Select 3 tabel	1	81340	81340	33780	33780	39380	39380
7	Select 4 tabel	1	93260	93260	39550	39550	44160	44160
8	Create table	4	4890	1222	20870	6956	6890	1745
9	Drop table	4	3460	865	4010	1002	4280	1070
10	Extended table create	12	31360	2613	38110	38110	38730	32227
11	Extended table drop	12	30480	2540	29160	2430	4280	1070
12	Extended site create	3	23070	7690	20870	6956	26150	8716
13	Extended site drop	3	9940	3313	5770	1923	11260	3753
	Total			233265		164315		185213

Pada tabel 6.3 dibandingkan penggunakan DBMS yang berbeda-beda secara bersamaan. Tiap kali perbandingan digunakan 3 DBMS yang mewakili 3 buah fragmentasi data. Di perbandingan pertama sebagai DBMS pusat digunakan Access, sedangkan sebagai DBMS lokal digunakan Access, Interbase, dan PostgreSQL. Pada perbandingan kedua sebagai DBMS pusat digunakan Access,

sedangkan sebagai DBMS lokal digunakan satu Interbase dan dua PostgreSQL. Pada perbandingan ketiga sebagai DBMS pusat digunakan PostgreSQL, sedangkan sebagai DBMS lokal Access, Interbase, dan PostgreSQL.

Pada empat buah tabel perbandingan berikutnya, pada sistem IDDB dilakukan fragmentasi data sebanyak dua buah, dan DBMS di pusat dan lokal menggunakan Access.

Pada tabel 6.4. dilakukan perbandingan dengan mengubah konfigurasi di bagian optimasi query digunakan optimasi mode semijoin, replicator, dan direct.

Tabel 6.4. Perbandingan dengan perubahan dibagian tipe optimasi

No	Perintah	Jumlah	Optimasi semijoin		Optimasi replicator		Optimasi direct	
			total	rata-rata	total	rata-rata	Total	rata-rata
1	Insert	30	52880	1762	54240	1808	54080	1802
2	Update	30	30740	1024	31170	1039	30070	1002
3	Delete	30	36530	2050	32760	1092	31920	1064
4	Select 1 tabel	4	16150	5150	16200	4050	13560	3390
5	Select 2 tabel	2	14390	7195	4670	2335	6320	3160
6	Select 3 tabel	1	8950	8950	3180	3180	2910	2910
7	Select 4 tabel	1	15320	15320	6980	6980	7360	7360
8	Create table	4	4170	1042	3940	985	3950	987
9	Drop table	4	3300	825	3580	895	3520	880
10	Extended table create	8	10220	1277	10490	1311	9570	1196
11	Extended table drop	8	8190	1023	8460	1057	8400	1050
12	Extended site create	2	7850	3925	9500	4750	8410	4205
13	Extended site drop	2	5990	2995	6370	3185	5880	2940
	Total			52538		32667		31946

Pada Tabel 6.5. dilakukan perbandingan dengan mengubah konfigurasi di bagian replikasi. Disini dibandingkan replikasi mode synchronous, asynchronous dengan time delay 10000 milisecond, dan tidak digunakan replikasi.

Tabel 6.5. Perbandingan dengan perubahan dibagian tipe replikasi

No	Perintah	Jumlah	Replicator syn		Replicator asyn		Replicator none	
			total	rata-rata	total	rata-rata	Total	rata-rata
1	Insert	30	52880	1762	33400	1113	28090	936
2	Update	30	30740	1024	36210	1207	28260	942
3	Delete	30	36530	2050	34030	1134	30700	1023
4	Select 1 tabel	4	16150	5150	18080	4520	17800	4450
5	Select 2 tabel	2	14390	7195	13620	6810	14940	7470
6	Select 3 tabel	1	8950	8950	8620	8620	9120	9120
7	Select 4 tabel	1	15320	15320	15100	15100	11700	11700
8	Create table	4	4170	1042	4500	1125	3180	795
9	Drop table	4	3300	825	3080	770	2800	700
10	Extended table create	8	10220	1277	11140	1392	9790	1223
11	Extended table drop	8	8190	1023	8230	1028	6540	817
12	Extended site create	2	7850	3925	7300	3650	9730	4865
13	Extended site drop	2	5990	2995	7190	3595	7250	3625
	Total			52538		50064		47666

Pada Tabel 6.6. dilakukan perbandingan dengan mengubah konfigurasi di bagian mode insert. Disini dibandingkan insert mode fragment, random, dan order.

Tabel 6.6. Perbandingan dengan perubahan dibagian tipe insert

No	Perintah	Jumlah	Insert fragment		Insert random		Insert order	
			total	rata-rata	total	rata-rata	Total	rata-rata
1	Insert	30	52880	1762	53510	1783	99030	3301
2	Update	30	30740	1024	47240	1574	31830	1061
3	Delete	30	36530	2050	56330	1877	33940	1131
4	Select 1 tabel	4	16150	5150	36810	9202	21650	5412
5	Select 2 tabel	2	14390	7195	16040	8020	14780	7390
6	Select 3 tabel	1	8950	8950	22800	22800	8620	8620
7	Select 4 tabel	1	15320	15320	27460	27460	12580	12580
8	Create table	4	4170	1042	4230	1057	5100	1275
9	Drop table	4	3300	825	3290	822	3180	795
10	Extended table create	8	10220	1277	9770	1221	15810	1976
11	Extended table drop	8	8190	1023	8570	1071	8460	1057
12	Extended site create	2	7850	3925	8730	4365	8730	4365
13	Extended site drop	2	5990	2995	4560	2280	6100	3050
	Total			52538		83532		52013

Pada Tabel 6.7. dilakukan perbandingan dengan mengubah konfigurasi di bagian pengolah transaksi. Disini dibandingkan pengolah transaksi mode

concurrent, single dan tidak adanya penggunaan transaksi manager.

Tabel 6.7. Perbandingan dengan perubahan dibagian tipe transaksi

No	Perintah	Jumlah	Trans concurrent		Trans single		Trans none	
			total	rata-rata	total	rata-rata	Total	rata-rata
1	Insert	30	52880	1762	54630	1821	55360	1845
2	Update	30	30740	1024	31540	1051	30780	1026
3	Delete	30	36530	2050	33480	1116	29770	992
4	Select 1 tabel	4	16150	5150	17190	4297	15380	3845
5	Select 2 tabel	2	14390	7195	14120	7060	14440	7220
6	Select 3 tabel	1	8950	8950	8620	8620	8290	8290
7	Select 4 tabel	1	15320	15320	14550	14550	12630	12630
8	Create table	4	4170	1042	3900	975	3790	947
9	Drop table	4	3300	825	2910	727.	3130	782
10	Extended table create	8	10220	1277	9840	1230	10090	1261
11	Extended table drop	8	8190	1023	8510	1063	8410	1051
12	Extended site create	2	7850	3925	9670	4835	9220	4610
13	Extended site drop	2	5990	2995	6650	3325	6860	3430
	Total			52538		50670		47929

Tabel 6.8. Perbandingan besar basis data (dalam byte).

Tabel	Kolom	Tipe	DBMS Pusat		DBMS Local 1		DBMS Local 2		DBMS Local 3	
			awal	akhir	Awal	Akhir	awal	akhir	awal	akhir
1	1	A	40960	65536						
	2	I	208896	282624						
	3	P	855117	897101						
	4	M	0	42194						
2	1	A,A	40960	395264	40960	235520				
	2	A,A,A	40960	413696	40960	188416	40960	186368		
	3	A,A,A,A	40960	413696	40960	186368	40960	186368	40960	186368
3	1	A,I,P,A	40960	413696	208896	287744	855117	921677	40960	186368
	2	A,I,P,P	40960	380928	208896	287744	855117	921677	855117	921677
	3	P,I,P,A	855117	1142861	208896	287744	855117	921677	40960	186368
4	1	A,A,A	40960	413696	40960	188416	40960	186368		
	2	A,A,A	40960	282624	40960	188416	40960	186368		
	3	A,A,A	40960	282624	40960	188416	40960	186368		
5	1	A,A,A	40960	413696	40960	188416	40960	186368		
	2	A,A,A	40960	413696	40960	188416	40960	186368		
	3	A,A,A	40960	251904	40960	188416	40960	186368		
6	1	A,A,A	40960	413696	40960	188416	40960	186368		
	2	A,A,A	40960	413696	40960	188416	40960	186368		
	3	A,A,A	40960	413696	40960	186368	40960	186368		
7	1	A,A,A	40960	413696	40960	188416	40960	186368		
	2	A,A,A	40960	413696	40960	188416	40960	186368		
	3	A,A,A	40960	413696	40960	188416	40960	186368		

Pada tabel 6.8. dilakukan perbandingan besarnya basis data yang digunakan, disini dicatat besar sebelum proses dan besar setelah proses. Tipe DBMS disini ialah A:Access, I:Interbase, P:PostgreSQL, M:MySQL.

6.2. Analisa Sistem IDDB

Pada bagian ini dilakukan analisa data-data hasil uji coba sistem. Analisa yang dilakukan analisa kompatibilitas, kecepatan, besar ruang penyimpanan, dan besar komunikasi data.

6.2.1. Analisa Kompatibilitas

Analisa ini digunakan untuk menguji kompatibilitas sistem ini, apakah bisa digunakan di segala platform dan DBMS yang merupakan salah satu tujuan dari perancangan sistem. Pengujian sistem IDDB dalam lingkungan Windows 95 sebagai berikut:

Sistem dapat berjalan dengan baik ketika dicoba dengan menggunakan DBMS Access, demikian pula ketika di gabung dengan PostgreSQL dan Interbase sehingga menghasilkan data pada tabel 6.3. Driver JDBC Interbase yaitu Interclient mempunyai bug sehingga ia tidak bisa digunakan sebagai DBMS pusat yaitu saat akses data secara bersamaan, tetapi jika ia digunakan sebagai DBMS lokal maka akan berjalan normal. DBMS MySQL tidak dapat digunakan karena driver JDBCnya mempunyai bug yaitu ia tidak mengembalikan type field hasil query, selain itu sering berhenti dan mengeluarkan pesan 'memory overflow',

sehingga DBMS ini digunakan sebagai pembanding saja. Pada PostgreSQL ada sedikit masalah yang mengganggu dimana ia mengenali karakter yang diawali '\' sebagai karakter khusus, seperti yang terdapat pada bahasa C atau Java, hal ini berpengaruh pada perintah 'extended site create' di parameter url database, sebagai contoh jika dimasukkan 'jdbc:interbase:d:\te.gdb' karena disana terdapat karakter '\' diikuti karakter 't' maka akan dikenali oleh PostgreSQL sebagai karakter tab, tetapi masalah ini dapat diatasi dengan menempatkan karakter '\' sebanyak dua kali.

Sedangkan saat penggunaan driver-driver ODBC milik Microsoft, melalui JDBC-ODBC bridge, ternyata hanya driver ODBC Access yang paling cocok untuk digunakan sebagai DBMS dalam sistem IDDB, sedangkan yang lain seperti driver untuk dBase dan FoxPro tidak bisa menerima tipe data integer, sedangkan driver untuk text tidak dapat mengembalikan tipe field dengan benar.

Ketika sistem IDDB dicoba di sistem operasi Linux, ia sering terhenti di bagian ServerSocket sehingga hubungan program aplikasi client tidak dapat masuk, ini merupakan bug yang ada JDK 1.1.1 di Linux, tetapi contoh program aplikasi client sistim informasi mahasiswa (dijelaskan di subbab berikutnya) dapat berjalan dengan baik di Linux maupun di Windows 95.

6.2.2. Analisa Kecepatan

Analisa kecepatan ini digunakan untuk menganalisa seberapa cepat kinerja sistem IDDB ini saat pengolahan data. Data-data untuk kebutuhan analisa ini dapat dilihat di tabel 6.1 sampai 6.7, disana dibandingkan kecepatan berbagai

sistem DBMS, dan sistem IDDB dengan berbagai perbedaan konfigurasi. Pada analisa kecepatan ini terdapat penyimpangan karena digunakan sistem operasi multitasking yaitu Windows 95, sering kali hasil pengukuran jika diulangi lagi hasilnya tidak sama persis, ini dikarenakan pada saat yang bersamaan sistem operasi memproses program lain.

Pada tabel 6.1. terlihat bahwa DBMS MySQL merupakan yang tercepat disusul oleh Access, PostgreSQL dan yang terakhir Interbase.

Pada tabel 6.2. terlihat bahwa total kecepatan penambahan 1 basis data fragmentasi mengakibatkan kecepatan menurun 50 %. Sedangkan pada operasi query select mengakibatkan kecepatan menurun sampai hampir 100 %. Sedangkan penambahan 1 tabel dalam operasi select mengakibatkan penurunan kecepatan sekitar 50 %.

Pada tabel 6.3 terlihat bahwa pemakaian DBMS yang berbeda mengakibatkan perbedaan kecepataan pula, semakin lambat DBMS yang dipakai maka kinerja sistem IDDB akan menjadi semakin lambat.

Pada tabel 6.4 terlihat bahwa kecepatan database yang menggunakan algoritma semijoin terlihat paling lambat disusul dengan optimasi tipe direct, dan terakhir tipe replicator. Tipe semijoin terlihat lambat sekali saat melakukan operasi select, jika ia dibandingkan dengan tipe yang lain operasi select mengalami kelambatan sekitar 200 % lebih lambat. Tipe direct yang seharusnya lebih lambat daripada tipe replicator (karena semua database berada di site pusat semua, tidak menghubungi DBMS lokal) terlihat lebih cepat, hal itu dikarenakan proses pengujian dilakukan pada site tunggal sehingga tidak perlu dilakukan

hubungan menggunakan jaringan komputer keluar.

Pada tabel 6.5. terlihat bahwa replicator tipe synchronous merupakan tipe yang paling lambat karena saat perintah query dilakukan dia selalu melakukan proses replikasi data, sedangkan saat tipe asynchronous proses agak lebih cepat, sedangkan jika replikasi data dimatikan maka akan terasa lebih cepat.

Pada tabel 6.6 terlihat bahwa insert tipe order merupakan tipe yang paling cepat, sedangkan tipe fragment lebih cepat daripada tipe random hal ini merupakan suatu penyimpangan yang telah disebutkan diatas, dimana yang seharusnya program tipe order atau random menghasilkan kecepatan yang lebih baik karena kode programnya lebih sedikit, tetapi disini terlihat lebih lambat, ini dikarenakan saat pengujian saat itu program yang lain sedang diaktifkan dan memori saat itu terlalu penuh sehingga proses swapping memory pada Windows 95 memperlambat hasil pengujian.

Pada tabel 6.7. terlihat bahwa jika pengatur transaksi dimatikan pemprosesan sistem IDDB menjadi lebih cepat, kemudian diikuti tipe single dan yang terakhir pengatur transaksi dihidupkan secara penuh yang akan mengakibatkan sistem menjadi lebih lambat.

6.2.3. Analisa Besar Ruang Penyimpanan

Analisa besar ruang penyimpanan ini digunakan untuk menganalisa berapa besar kebutuhan sistem IDDB untuk ukuran ruang penyimpanan data. Data-data untuk kebutuhan analisa ini dapat dilihat di tabel 6.8, disana dibandingkan semua besar basis data selesai proses pengujian pada sistem IDDB.

Pada tabel bagian no1.terlihat bahwa basis data yang dibuat oleh tiap-tiap DBMS lebih kecil besarnya daripada hasil penggabungan DBMS, malah pada DBMS MySQL besar sebelum proses 0 byte, ini dikarenakan ia belum membuat suatu struktur data apapun, struktur data hanya dibuat kalau tabel telah ada, sedangkan DBMS PostgreSQL memerlukan tempat penyimpanan data paling besar.

Pada tabel 6.8. bagian no 2. Terlihat bahwa semakin sedikit jumlah fragmentasi, semakin sedikit jumlah total data untuk menyimpan basis data. Ini terlihat bahwa jumlah data pada basis data yang terfragmentasi satu bagian lebih sedikit daripada dua bagian demikian pula dari yang tiga bagian.

Pada tabel 6.8. bagian no 3. Besar tempat penyimpanan untuk setiap basis data fragmentasi selalu tetap besarnya.

Pada tabel 6.8. bagian no 4. Terlihat operasi semijoin membutuhkan tempat penyimpanan lebih besar daripada operasi optimasi bertipe direct dan replicator, ini dikarenakan pada saat semijoin sistem membuat dan menghapus tabel secara temporari.

Pada tabel 6.8. bagian no 5. Terlihat jika replicator dimatikan besar data bisa dikurangi hampir separuhnya, ini dikarenakan tidak perlu dibuat tabel dan disimpan data replikasi pada sistem IDDB.

Pada tabel 6.8. bagian no 6. dan 7 Terlihat bahwa perubahan pada tipe insert dan transaksi tidak mempengaruhi besar penyimpanan data pusat. Tetapi pada tipe insert random atau order mempengaruhi besar basis data fragmentasi, tipe random bisa membuat besar basis data fragmentasi lebih besar dari yang lain,

sedangkan tipe order membuat besar basis data fragmentasi sama, karena terdistribusi sama rata. Sedangkan pada perubahan tipe transaksi tidak mempengaruhi besar basis data fragmentasi, karena tidak ada data yang disimpan berhubungan dengan proses transaksi.

6.2.4. Analisa Besar Komunikasi Data

Bagian ini menganalisa besarnya data yang ditransfer selama proses perintah query yang lewat jaringan komputer dari DBMS lokal ke bagian server IDDBMaster. Perintah query yang paling banyak membutuhkan proses transfer data ialah perintah select yang selain mengembalikan tupel-tupel hasil perintah query dalam prosesnya ia membutuhkan transfer data yang cukup besar, sedangkan perintah yang bertipe update hanya mengembalikan jumlah tupel saja, oleh sebab itu analisa hanya dilakukan pada perintah query select. Perbedaan konfigurasi di bagian tipe optimasi paling berpengaruh pada proses pada perintah query tersebut, sehingga disini dibandingkan besarnya data selama komunikasi di ketiga tipe optimasi yaitu semijoin, direct dan replicator. Yang dihitung pada analisa ini ialah besar data tabel yang ditransfer selama proses query, tidak termasuk data kontrol saat komunikasi data antar DBMS dilakukan. Disini perintah query seperti create dan drop table tidak diperhitungkan, karena tidak terlalu besar data yang ditransfer.

Rumus penghitungan sebagai berikut :

Banyaknya data = (total jumlah data dalam 1 tupel) x banyaknya tuple

Data-data contoh untuk analisa sebagai berikut :

Struktur data tabel :

* mahasiswa (nrp char(10), nama char(30), kodejur char(3)) 30 tupel
* kuliah (kode char(6), nama char(30), kodejur char(3)) 15 tupel
* frs (nrp char(10), kode char(6), kodejur char(3)) 30 tupel
* jurusan (kodejur char(3), nama char(20)) 3 tupel

Operasi select :

1. select * from mahasiswa, hasil 30 tupel
2. select * from kuliah, hasil 15 tupel
3. select * from frs, hasil 30 tupel
4. select * from jurusan, hasil 3 tupel
5. select mahasiswa.nrp, mahasiswa.nama, frs.kode from mahasiswa, frs where mahasiswa.nrp=frs.nrp, hasil 30 tupel
6. select kuliah.kode, kuliah.nama, frs.nrp from kuliah, frs where kuliah.kode = frs.kode, hasil 30 tupel
7. select mahasiswa.nrp, mahasiswa.nama, kuliah.kode, kuliah.nama from mahasiswa, kuliah, frs where mahasiswa.nrp=frs.nrp and kuliah.kode=frs.kode, hasil 30 tupel

8. select jurusan.kodejur, jurusan.nama, mahasiswa.nrp, mahasiswa.nama, kuliah.kode, kuliah.nama from mahasiswa, kuliah, frs, jurusan where mahasiswa.nrp = frs.nrp and kuliah.kode = frs.kode and mahasiswa.kodejur = jurusan.kodejur, hasil 30 tupel

Data fragmentasi di 2 buah DBMS lokal. Disini operasi query create dan drop table tidak diperhitungkan.

Pada bagian berikut dianalisa besar data komunikasi untuk tiap-tiap operasi select diatas.

Optimasi tipe semijoin :

1. Data yang diambil dalam 1 tupel besarnya 1 tupel penuh, yang merupakan jumlah semua besarnya field dan disini tidak dikenakan pengurangan dari hasil operasi semijoin. Jadi besar data = (10 + 30 + 3) x 30 = 43 x 30 = 1290 byte.
2. Data yang diambil dalam 1 tupel besarnya 1 tupel penuh, yang merupakan jumlah semua besarnya field dan disini tidak dikenakan pengurangan dari hasil operasi semijoin. Jadi besar data = (6 + 30 + 3) = 39 x 15 = 585 byte.
3. Data yang diambil dalam 1 tupel besarnya 1 tupel penuh, yang merupakan jumlah semua besarnya field dan disini tidak dikenakan pengurangan dari hasil operasi semijoin. Jadi besar data = (10 + 6 + 3) x 30 = 19 x 30 = 570 byte.
4. Data yang diambil dalam 1 tupel besarnya 1 tupel penuh, yang merupakan jumlah semua besarnya field dan disini tidak dikenakan pengurangan dari hasil operasi semijoin. Jadi besar data = (3 + 20) x 3 = 23 x 3 = 69 byte.
5. Urutan langkah-langkah hasil algoritma semijoin yang dilakukan untuk DBMS

lokal sebagai berikut :

- select frs.kode, frs.nrp from frs
- insert into tmpfrs values(frs.nrp) (frs=30 tupel dikirim ke 2 site)
- select mahasiswa.nrp, mahasiswa.nama from mahasiswa, tmpfrs where mahasiswa.nrp=tmpfrs.nrp

besar data untuk perintah select1 = (6+10)*30 = 16 x 30 = 480 byte

besar data untuk perintah insert = 10 x 30 x 2=600 byte

besar data untuk perintah select2 = (10 + 30) x 30= 1200 byte

Jadi total jumlah data = 480 + 600+1200=2280 byte

6. Urutan langkah-langkah hasil algoritma semijoin yang dilakukan untuk DBMS lokal sebagai berikut :

- select frs.kode, frs.nrp from frs
- insert into tmpfrs values(frs.nrp) (frs=30 tupel dikirim ke 2 site)
- select kuliah.kode, kuliah.nama from kuliah, tmpfrs where kuliah.kode=frs.kode

besar data untuk perintah select1 = (6+10)*30 = 16 x 30 = 480 byte

besar data untuk perintah insert = 10 x 30 x 2=600 byte

besar data untuk perintah select2 = (6 + 30) x 15= 540 byte

Jadi total jumlah data= 480 + 600+540=1720 byte

7. Urutan langkah-langkah hasil algoritma semijoin yang dilakukan untuk DBMS lokal sebagai berikut :

- select mahasiswa.nrp, mahasiswa.nama from mahasiswa
- insert into tmpmahasiswa values(mahasiswa.nrp) (frs=30 tupel dikirim ke 2 site)
- select kuliah.kode, kuliah.nama from kuliah
- insert into tmpkuliah values(kuliah.kode) (frs=30 tupel dikirim ke 2 site)
- select frs.nrp, frs.kode from frs, tmpmahasiswa, tmpkuliah where frs.kode=tmpkuliah.kode and frs.nrp=tmpmahasiswa.nrp

  besar data untuk perintah select1 = (10+30)*30 = 40 x 30 = 1200 byte

  besar data untuk perintah insert1 = 10 x 30 x 2=600 byte

  besar data untuk perintah select2 = (6 + 30) x 15= 540 byte

  besar data untuk perintah insert2 = 6 x 15 x 2=180 byte

  besar data untuk perintah select3 = (10+6) x 30= 480 byte

  Jadi total besar data = 1200+600+540+180+480=3000 byte

8. Urutan langkah-langkah hasil algoritma semijoin yang dilakukan untuk DBMS lokal sebagai berikut :

- select jurusan.kodejur, jurusan.nama from jurusan
- insert into tmpjurusan values(jurusan.kodejur)
- select kuliah.kode, kuliah.nama from kuliah
- insert into tmpkuliah values(kuliah.kode)
- select frs.nrp, frs.kode from frs, tmpkuliah where frs.kode=tmpkuliah.kode

- insert into tmpfrs values(frs.nrp, frs.kode)
- select mahasiswa.nrp, mahasiswa.nama, mahasiswa.kodejur from mahasiswa, tmpfrs, tmpjurusan where mahasiswa.kodejur = tmpjurusan.kodejur and mahasiswa.nrp = tmpfrs.nrp

besar data untuk perintah select1 = (3+20)*3 = 23 x 3 = 69 byte

besar data untuk perintah insert1 = 3 x 3 x 2=18 byte

besar data untuk perintah select2 = (6 + 30) x 15= 540 byte

besar data untuk perintah insert2 = 6 x 15 x 2=180 byte

besar data untuk perintah select3 = (10+6) x 30= 480 byte

besar data untuk perintah insert3 = (10 + 6) x 30 x 2=16 x 30 x 2 = 960 byte

besar data untuk perintah select4 = (10 + 30 + 3) x 30= 1290 byte

Jadi total jumlah data = 69+18+540+180+480+960+1290=3537 byte

Catatan untuk langkah-langkah penggunaan query hasil algoritma semi-join dapat dilihat pada lampiran D.

Optimasi tipe direct :

Besar data yang diambil ke tiap-tiap DBMS lokal pada no 1 sampai 4 sama dengan operasi tipe semijoin, karena semua field diambil seluruhnya, tidak dikenakan operasi semijoin.

5. Besar data = (10 + 30 + 6) x 30 = 46 x 30 = 1380 byte

6. Besar data = (6 + 30 + 10) x 30 = 46 x 30 = 1380 byte

7. Besar data = (10 + 30 + 6 + 30) x 30 = 76 x 30 = 2280 byte

8. Besar data = (3 + 20 + 10 + 30 + 6 + 30) x 30 = 99 x 30 = 2970 byte

Optimasi tipe replicator :

Tipe ini tidak memerlukan hubungan komunikasi keluar lewat jaringan komputer karena data replikasi terletak di DBMS pusat sendiri, tetapi untuk kelipatan waktu tertentu ia melakukan proses pembandingan data replikasi dengan data di DBMS lokal jika tipe asynchronous, sedangkan jika tipe synchronous setiap kali melaksanakan perintah query baru dilakukan proses pembandingan.

Proses pembandingan dilakukan dengan rumus sebagai berikut :

Jumlah data = (jumlah tabel x jumlah site) x (ukuran bilangan counter yang digunakan untuk pembanding) 4 byte,

Jika terjadi update data :

Jumlah data = (jumlah tabel x jumlah site) x jumlah perubahan data tiap DBMS lokal byte

jumlah perubahan data tiap DBMS lokal = jumlah field yang belum diupdate x (besar 1 tupel pada iddblog)318 byte

Jadi pada contoh data diatas jika tidak terjadi update data, jumlah data

yang ditransfer ialah :

Jumlah data = (4 x 2) x 4 = 32 byte.

Dari sini disimpulkan penggunaan tipe replikator lebih sedikit mengadakan komunikasi lewat jaringan komputer untuk mengakses data di DBMS lokal, sedangkan tipe direct mengakses data besarnya tepat sama dengan hasil keluaran sistem. Tipe semijoin efektif jika hanya sedikit jumlah field yang dibutuh dari jumlah keseluruhan field tabel yang akan dibaca dan jumlah tupel yang dihasilkan lebih sedikit daripada jumlah tupel keseluruhan.

6.3. Contoh Program Aplikasi Client: Sistem Informasi Mahasiswa

Pada bagian ini dibahas mengenai sebuah aplikasi penggunaan sistem basis IDDB ini dengan memanfaatkan fasiltas terdistribusnya data. Sistem IDDB ini digunakan untuk mengolah sistem informasi mahasiswa.

6.3.1. Desain Sistem

Sistem informasi mahasiswa yang dipakai mempunyai keterangan sebagai berikut :

* Mahasiswa pasti berada di salah satu jurusan. Tiap-tiap jurusan pasti mempunyai mahasiswa.

* Tiap-tiap mata kuliah berada di salah satu jurusan. Tiap-tiap jurusan pasti mempunyai banyak mata kuliah.

* Mahasiswa dapat mengambil lebih dari satu mata kuliah. Tiap-tiap mata kuliah pasti diambil oleh mahasiswa.

Sistem informasi ini meminta agar tiap-tiap data dapat diakses menurut jurusan masing-masing dan transfer data dilakukan olah jaringan komputer lokal tiap-tiap jurusan untuk menghemat biaya komunikasi, sedangkan pada saat tertentu dapat diakses data secara keseluruhan atau global, dimana semua data yang ada di tiap-tiap jurusan dapat digabung dan ditampilkan langsung ke pemakai.

Bentuk perancangan diagram ER sistem informasi mahasiswa ini dapat dilihat di Gambar 6.1. Dari diagram ER tersebut dapat dibuat tabel-tabel basis data untuk sistem informasi mahasiswa ini dengan penyesuaian yang dilakukan agar dapat memanfaatkan fasilitas terdistribusinya data pada sistem IDDB ini.

Struktur data tabel sistem informasi mahasiswa :

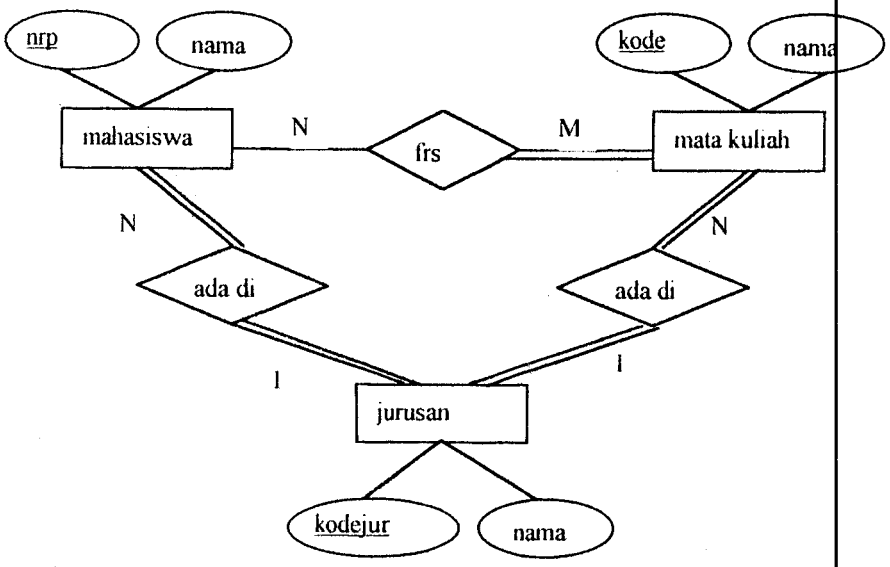
mahasiswa (nrp, nama, kodejur)

kuliah (kode, nama, kodejur)

jurusan (kodejur, nama)

frs (nrp, kode, kodejur)

Melalui penggunaan sistem IDDB ini, tabel-tabel yang ada dipecah menurut jurusan masing-masing. Pemecahan atau fragmentasi tabel-tabel yang ada dapat dilakukan dengan memberi aturan fragmentasi pada tabel tersebut yang membedakan tupel-tupel data yang ada untuk ditempatkan di suatu DBMS lokal tertentu. Karena alasan itu field kodejur yang ada di tiap-tiap tabel digunakan

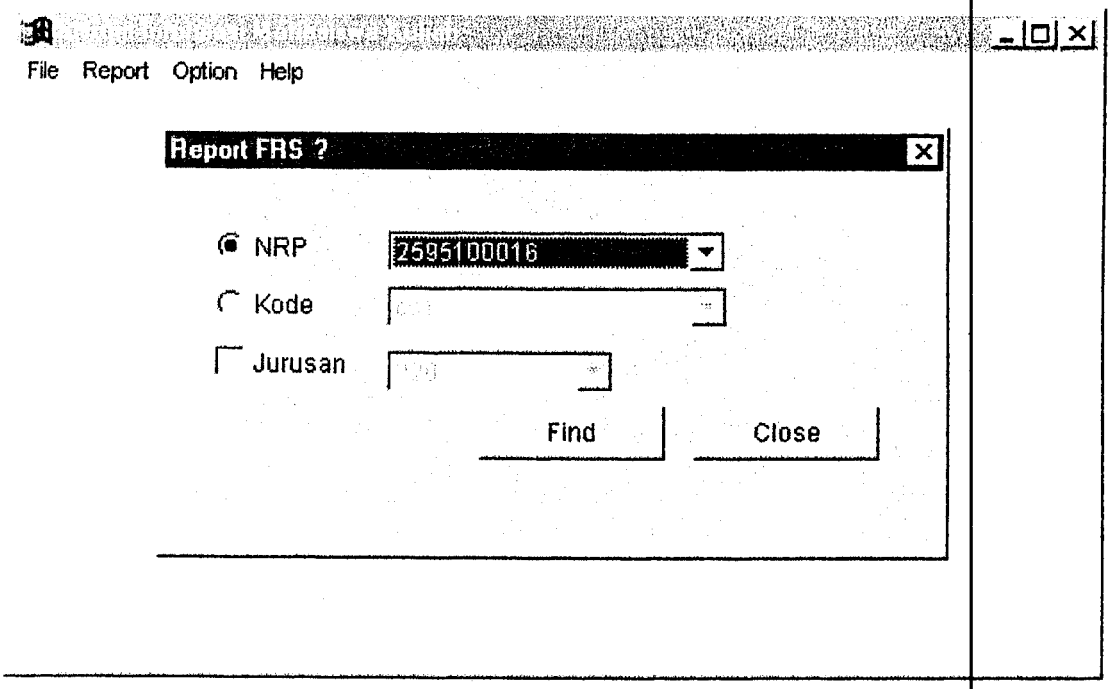


Gambar 6.1. Diagram ER Sistem Informasi Mahasiswa untuk IDDB

sebagai aturan fragmentasi. Sedangkan pada tabel frs yang semestinya tidak ada field kodejur, ditambahkan field kodejur, (yang pengisiannya diambil dari field kodejur tabel mahasiswa), hal ini karena sistem IDDB, hanya dapat menggunakan aturan fragmentasi data dengan field-field dari tabel itu sendiri, tidak dapat mengambil data dari field-field milik tabel yang lainnya.

6.3.2. Penjelasan Progam

Program aplikasi client ini (gambar 6.2.) dibuat menggunakan bahasa Java agar dapat menggunakan driver IDDB JDBC, atau driver JDBC yang lain caranya dengan menjalankan program ini dengan parameter tambahan nama file konfigurasi. File konfigurasi tersebut berisi class driver JDBC tersebut, letak database, user dan password. Jadi sifat kemudahan untuk mengganti-ganti driver sesuai dengan kebutuhan (portabilitas)diimplementasikan dalam program aplikasi ini tanpa harus merubah kode program dan melakukan kompilasi program.



Gambar 6.2. Sistem Informasi Mahasiswa

Program aplikasi ini secara automatis mengenali apakah driver yang digunakan itu merupakan driver JDBC IDDB atau bukan, kalau bukan maka pada menu Option akan terdapat menu item create table dan drop table yang gunanya sebagai inisialisai membuat tabel-tabel yang diperlukan oleh sistem informasi mahasiswa, sedangkan drop table untuk menghapusnya.

Jika dikenali driver JDBC IDDB maka diperiksa apakah mode global atau lokal. Pada mode global yang berarti pemakai tersebut bisa mengakses data-data yang berada di semua jurusan, pada menu option akan keluar menu item 'extended create table dan 'extended drop tabel' yang gunanya sebagai inisialisasi membuat tabel dan aturan fragmentasi, sedangkan yang satunya untuk menghapusnya. Jika ia dalam mode lokal, ia hanya bisa mengakses data-data disalah satu jurusan tertentu sesuai yang data di file konfigurasi JDBC IDDB.ini maka menu option tidak akan keluar dan menu item jurusan pada menu file tidak

dapat diakses. Untuk mengubah mode global atau lokal pada file JDBC IDBB.ini bagian lokal diberi nilai 'on' untuk menjadi mode lokal, sedangkan jika diberi nilai 'off' maka ia akan menjadi mode global. Pemilihan mode ini akan nampak saat masuk ke menu 'help' menu item 'about', disana terdapat informasi driver yang digunakan dan versinya.

Pada saat mode lokal setiap kali penambahan data mahasiswa, mata kuliah, atau frs tidak dapat dilakukan pemilihan item jurusan di ubah ke jurusan lain, ini untuk menjaga bahwa data yang dimasukkan sesuai dengan aturan fragmentasi untuk lokal DBMS saat itu.

6.3.3. Evaluasi Sistem

Saat dilakukan perubahan data yang dilakukan pada program mode global, akan terlihat langsung oleh program mode lokal. Sedangkan perubahan yang dilakukan oleh program mode lokal akan terlihat langsung juga oleh program mode global jika IDDBMaster, tetapi jika bagian replikasi diset tipe asynchronous dan optimasi diset tipe replikasi maka perubahan pada data lokal akan terlihat setelah jangka waktu tertentu menurut besarnya waktu update replikasi asynchronous pada IDDBMaster.

Program aplikasi ini lebih cepat jika mode pengaksesan data ke sistem IDDB dilakukan secara lokal, kecepatannya mirip saat digunakan driver JDBC dari database lain yang mengakses data tanpa fragmentasi. Sedangkan pada mode global kecepatan menurun sekitar enam kali lebih lambat dari pada mode lokal sesuai analisa kecepatan pada subbab sebelumnya.

BAB VII

KESIMPULAN DAN SARAN

Bab ini menjelaskan kesimpulan dan saran Tugas Akhir ini. Kesimpulan dan saran diambil berdasarkan studi literatur, perancangan, dan hasil uji coba perangkat lunak yang telah diuraikan dalam bab-bab sebelumnya.

7.1. Kesimpulan

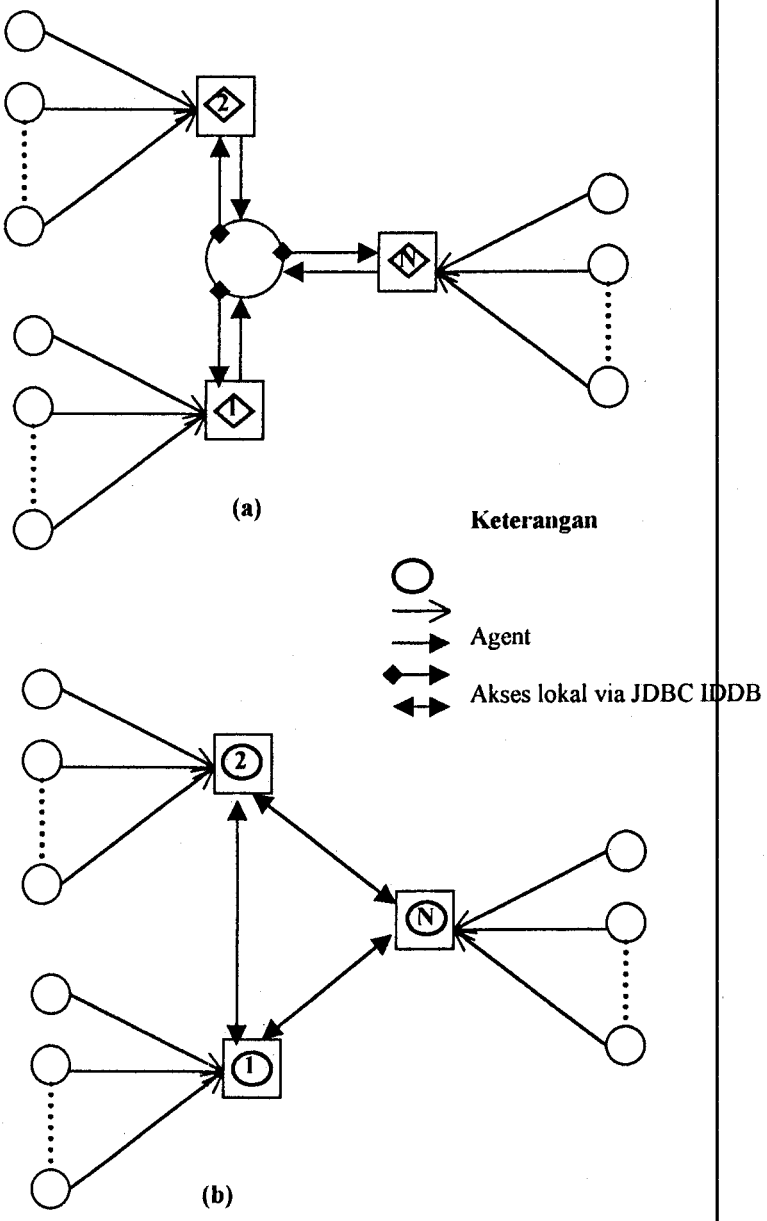
- Sistem IDDB ini merupakan sistem manajemen basis data terdistribusi yang mempunyai jenis arsitektur tipe heterogen. Sedangkan jika dipandang secara non-arsitektural memiliki otonomi lokal untuk membaca dan mengubah data secara lokal dan memiliki tipe replikasi sebagian (*partial replication*).
- Penggunaan Java dan JDBC mengakibatkan program sistem IDDB ini dapat dijalankan di berbagai *platform* yang berbeda perangkat keras dan sistem operasi serta dengan DBMS yang berbeda-beda.
- Akses baca data melalui operasi *query* select, mempunyai waktu pemprosesan yang paling cepat jika menggunakan data replikasi, dibandingkan jika menggunakan optimasi semijoin. Hal ini dikarenakan

oleh proses manipulasi data yang hanya terjadi pada bagian *server* di pusat.

- Penggunaan optimasi semijoin akan menjadi lebih efektif jika jumlah tupel hasil *query* lebih sedikit dibandingkan jumlah tupel pada tiap-tiap tabel yang digunakan dalam proses *query*. Proses pembuatan tabel sementara dan penghapusan tabel yang ada pada langkah-langkah *query* hasil operasi semijoin ternyata memperlambat proses operasi *query* select yang diberikan.
- Proses sikronisasi antara data replikasi dan data utama pada tiap-tiap DBMS lokal dapat dilakukan dengan baik. Kecepatan proses ini tergantung pada banyaknya data, dimana jika terdapat banyak data yang harus direplikasi, maka proses yang terjadi akan berjalan lebih lambat.

7.2. Saran

- Arsitektur sistem manajemen basis data terdistribusi yang telah dibuat dapat dikembangkan lebih lanjut, dimana rancangan perubahan arsitektur dapat dilihat pada gambar 7.1. Dalam gambar 7.1a, arsitekur ini masih menggunakan *primary site* sebagai koordinator, tetapi modul Remote Agent dipecah dari *primary site* menjadi program yang terpisah dan dijalankan di tiap-tiap *site* lokal. Manfaat dari pemecahan ini adalah dimungkinkannya penanganan langsung permintaan data oleh *client* baik secara lokal maupun secara global (melalui *primary site*), sehingga, tidak terjadi kemungkinan kesalahan letak sebuah tupel data pada DBMS lokal,



Gambar 7.1 Rancangan perubahan arsitektur IDDB

dan lalulintas komunikasi data akan lebih terjaga. Untuk mengakses data program aplikasi, *client* melakukan hubungan dengan Remote Agent langsung menggunakan *driver* JDBC bagian Remote Agent.

- Kemungkinan pengembangan berikutnya seperti ditunjukkan pada gambar 7.1b, Remote Agent berubah menjadi Agent-Agent yang tidak mempunyai

suatu *primary site*, sehingga benar-benar terdistribusi.

- Optimasi saat pelaksanaan *query* dilaksanakan lebih lengkap misalnya dengan dilakukan optimasi persamaan *query* sesuai dengan prinsip aljabar relasional, lalu proses optimasi pada perintah *query* dengan melihat aturan fragmentasi tabel, sehingga cukup menghubungi *site-site* yang termasuk dalam aturan fragmentasi saja saat melakukan operasi tersebut tidak perlu menghubungi semua *site* seperti yang dilakukan dengan cara saat ini.
- Dilakukan kompresi saat pengiriman data sehinga beban komunikasi dapat diperringan, serta dilakukan enkripsi dengan metoda DES terhadap pengiriman data dan *password* agar tidak mudah diketahui orang (*hacker*).
- Adanya kemampuan untuk memindah data dari suatu *site* fragmentasi ke *site* yang lain, dan merubah aturan fragmentasi, serta kemampuan untuk merubah struktur data tabel.

DAFTAR KEPUSTAKAAN

1. Alfred Aho, Ravi Sethi and Jeffrey Ullman, "***Compilers: Principles,Techniques and Tools***", Addison-Wesley, (1986).
2. Atre S. "***Distributed Database, Cooperative Processing, & Networking***", McGraw-Hill, (1993).
3. Bell D.A. and Grimson J.B., "***Distributed Database Systems***", Addison-Wesley, (1992).
4. Ceri S. and Pelagatti G, "***Distributed Databases: Principles and Systems***", New York: McGraw-Hill, (1984).
5. Elmasri R. and Navathe S.B., "***Fundamentals of Database Systems***", Benyamin/Cummings, (1994).
6. Jackson J.R. and McClellan A.L., "***Java by Example Edisi Indonesia***", Penerbit Andi Yogyakarta, (1996).
7. Stevens W.R., "***Unix Network Programming***", Prentice Hall, (1990).
8. Stonebraker M, Aoki P.M, Devine R, Litwin W, and Olson M, "***Mariposa: A New Architecture for Distributed Data***", University of California at Berkeley, (1994).
9. Stonebraker M, Aoki P.M, Litwin W, Pfeffer Avi, Sidell J, Sah A, Staelin Carl, and Yu A, "***Mariposa: a wide-area distribute database system***", The VLDB Journal, Springer-Verlag, (1996) 5: 48-63.
10. Yu A. and Chen J., "***The Postgres95 User Manual***", University of California at Berkeley, (1995).
11. "***Communication Procotol between the Frontend and POSTGRES Backend***", University of California at Berkeley, (1992).
12. "***InterClient Documentation***", Interbase Corp, (1997).
13. "***JavaCC Documentation***", Sun Microsystems, Inc., (1997).

14. "***Java Development Kit 1.1.3 Documentation***", Sun Microsystems, Inc., (1997).

15. "***JDBC : A Java SQL API***", Sun Microsystems, Inc., (1997).

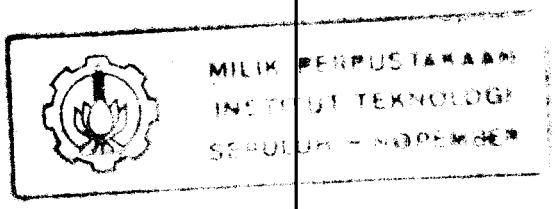
16. "***MARIPOSA Distributed Database Management System***", University of California at Berkeley, (1996).

17. "***ODBC 3.0 Reference***", Microsoft Corporation, (1996).

18. "***Postgres Architecture and Historical Lore***", University of California at Berkeley, (1992).

19. "***(Second Informal Review Draft) ISO/IEC 9075:1992, Database Language SQL- July 30, 1992***", Digital Equipment Corporation, (1992).

20. "***The JDBC (tm) API Version 1.20***",Sun Microsystems, Inc., (1997).



LAMPIRAN A

GRAMMAR SQL MINIMUM

```
CREATE TABLE table_name "("
        col_name        col_type [ NOT NULL [ PRIMARY KEY ]]
        (","col_name  col_type [ NOT NULL [ PRIMARY KEY ]]) * ")"
col_type CHAR["("num")"] | INTEGER | FLOAT


DROP TABLE table_name

INSERT INTO table_name [(column (","column)*)]
VALUES"("value (","value)*")"

DELETE FROM table_name
[WHERE column OPERATOR value
        ((AND | OR )column OPERATOR value)*]

SELECT [table"."]column (","[table"."]column)*
FROM table [= alias] (","table[=alias])*
[WHERE [table"."] column OPERATOR VALUE
((AND | OR) [table"."]column OPERATOR VALUE)*]
[ORDER BY [table"."]column [ASC|DESC](","[table"."]column [ASC|DESC])*]

UPDATE table_name SET column=value (","column=value)*
WHERE column OPERATOR value
((AND|OR )column OPERATOR value)*

OPERATOR <,>,=,<=,>=,<>

VALUE literal value atau column name
```

LAMPIRAN B

STRUKTUR DATA BASIS DATA UTAMA

```
IDDBUSER (
        iuser char(20),
        ipassword char(50),
        ilevel integer)

IDDBSITE (
        isiteid char(20),
        idriver char(50),
        iurl char(100),
        iuser char(20),
        ipassword char(50))

IDDBTB(
        itbname char(20),
        iowner char(20))

IDDBTBSITE(
        itbname char(20),
        isiteid char(20),
        ifname char(20),
        ifrop integer,
        ifrbool integer,
        ifrtype integer,
        ifrintvalue integer,
        ifrfloatvalue float,
        ifrcharvalue char(50))

IDDBTBFIELD(
        itname char(20),
        ifname char(20),
        iftype integer,
        iflength integer,
        ifnotnull integer,
        ifpkey integer,
        ino integer)

IDDBOPTION(
        ioption char(30),
        invalue integer,
        icvalue char(100))

IDDBLOG (                    *Digunakan juga untuk log tiap-tiap table dengan nama log table_name.
        idate char(30),
        icounter integer,
        ioperation char(500)
        itype integer,
        iuser char(20),
        isiteid char(20))

IDDBTBCOUNT (                *Digunakan di lokal dan di pusat.
        itbname char(20),
        isiteid char(20),
        icounter integer)
```

LAMPIRAN C

PROTOKOL KOMUNIKASI

Karakter Awal	Penjelasan	
O	Server Ok	: akhir dari suatu prosess
E	Error	: ada kesalahan
W	Warning	: peringatan
Q	Query	: perintah query
S	Shutdown connection	: akhiri koneksi socket
D	Data transfer	: transfer data tuple-tupe
T	meTa data transfer	: transfer tipe data tuple-tuple
C	Command	: perintah
U	Update Count	: jumlah query update
X	eXtended	: data tambahan
A	Autocommit	: commit secara automatis
L	transactionLevel	: level transaksi

LAMPIRAN D

CONTOH LANGKAH-LANGKAH OPERASI SEMI-JOIN PADA BASIS DATA TERFRAGMENTASI HORISONTAL

CONTOH 1:

STRUKTUR DATA BASIS DATA A & B :

A :
    x
    y

B :
    x
    z

PERINTAH :

```
select A.x, A.y, B.z from A, B where A.x=B.x
```

LANGKAH-LANGKAH :

PRIMARY SITE	**REMOTE SITE**
create table tmpA(x,y)	
	*create table tmpA(x)
	*select A.x, A.y from A
+insert into tmpA values(x,y)	
	+*insert into tmpA values(x)
create table tmpB(x, z)	
	*select B.x, B.z from tmpA, B
where tmpA.x=B.x	
+insert into tmpB values(x,z)	
select tmpA.x, tmpA.y, tmpB.z from tmpA, tmpB where tmpA.x=tmpB.x	
drop table tmpA	
	*drop table tmpA
drop table tmpB	

CONTOH 2

STRUKTUR DATA BASIS DATA A, B, & C :

A :
- m
- n
- o

B :
- m
- x
- y

C :
- n
- a
- b

PERINTAH :

```
select A.o, B.m, B.x, C.b from A, B, C where A.m=B.m and A.n=C.n
```

LANGKAH-LANGKAH :

PRIMARY SITE	REMOTE SITE
create table tmpB(m,x)	
	*create table tmpB(m)
	*select B.m, B.x from B
+insert into tmpB values(m,x)	
	+*insert into tmpB values(m)
create table tmpC(n,b)	
	*create table tmpc(n)
	*select C.n, C.b from C
+insert into tmpC values(n,b)	
	+*insert into tmpC values(n)
create table tmpA(m,n,o)	
	*select A.m, A.n, A.o from A, tmpB, tmpC where A.m=tmpB.m and A.n=tmpC=n
+insert into tmpA values(m,n,o)	
select tmpA.o, tmpB.m, tmpB.x, tmpC.b from tmpA, tmpB, tmpC where tmpA.m=tmpB.m and tmpA.n=tmpC.n	
drop table tmpA	
	*drop table tmpB
drop table tmpB	
	*drop table tmpC
drop table tmpC	

Keterangan tanda :

+ : perintah ini dikerjakan berulang-ulang sebanyak nilai baris yang ada

** : perintah ini dikerjakan di semua remote site yang ada.*

LAMPIRAN E

PESAN KESALAHAN

1001, IDDBMaster Error : In MainManager, Exception: XXX
1002, IDDBMaster Error : File configuration IDDBMaster.ini: optimizer = replicator, but replicator = off !"),
1003, IDDBMaster Error : While check existing MAIN table catalog IDDB, Exception: XXX
1004, IDDBMaster Error : While Initialisation Connection to MAIN DBMS, Exception: XXX
1005, IDDBMaster Error : While Try to load class Driver JDBC MAIN DBMS, Exception: XXX
1006, IDDBMaster Error : Fail in close, Exception: XXX

1101, Parser Error : Encounter Error during parsing, Exception: XXX

1201, User Authentication Error : Initialisation Fail, Exception: XXX
1202, User Authentication Error : Close Fail, Exception: XXX
1203, User Authentication Error : While find data user: XXX and password from 'iddbuser', Exception: XXX

1301, Transaction Manager Error : Initialisation Fail, Exception: XXX
1302, Transaction Manager Error : BLOCKED by operation 'update: XXX' from Transaction No: # # 'XXX' ROLLBACK READ this transaction: #, because will not be commited
1303, Transaction Manager Error : This query was done before Transaction No: # Commit: # 'XXX' ROLLBACK READ this transaction : #, because tsNow < tsUpdate, will not be commited
1304, Transaction Manager Error : BLOCKED by operation 'update: XXX' from Transaction No: # # 'XXX' ROLLBACK UPDATE this transaction: #, because will not be commited
1305, Transaction Manager Error : COMMIT 'XXX' ROLLBACK UPDATE this Transaction No: #, because tsNow < tsUpdate, will not be commited
1306, Transaction Manager Error : COMMIT 'XXX' ROLLBACK UPDATE this Transaction No: #, because tsNow < tsRead, will not be commited
1307, Transaction Manager Error : Encounter Error during TransactionManager Proccess, Exception: XXX
1308, Transaction Manager Error : Encounter Error during write log, Exception: XXX
1309, Transaction Manager Error : Transaction Single mode, wait for next transaction
1310, Transaction Manager Error : Query: XXX ROLLBACK READ this Transaction No: #, because tsNow < tsUpdate, will not be commited
1311, Transaction Manager Error : Query: XXX ROLLBACK UPDATE this Transaction No: #, because tsNow < tsUpdate, will not be commited
1312, Transaction Manager Error : Query: XXX ROLLBACK UPDATE this Transaction No: #, because tsNow < tsRead, will not be commited

1401, Connector Error : While initialisation, Exception: XXX
1402, Connector Error : User: XXX not found !
1403, Connector Error : Password not valid !
1404, Connector Error : Database: XXX not valid, always use database: 'iddb'
1405, Connector Error : Connector No : # abnormal out !, Exception: XXX
1406, Connector Error : While send Tupels, Exception: XXX
1407, Connector Error : While send Header of Tuples, Exception: XXX
1408, Connector Error : While send Result, Exception: XXX
1409, Connector Error : While send count of Updated Tuples, Exception: XXX
1410, Connector Error : While send Error, Exception: XXX
1411, Connector Error : While send Warning, Exception: XXX

1501, Remote Agent Error : Initialisation Fail, Exception: XXX
1502, Remote Agent Error : Close Fail, Exception: XXX
1503, Remote Agent Error : Queue RemoteAgent, siteid: XXX Full !
1504, Remote Agent Error : Exception while running, siteid: XXX veQueryNo: XXX query: XXX Exception: XXX

1601, Replicator Error : Running sleep Thread ! Exception: XXX
1602, Replicator Error : Different operation comparation log number ! : TbCount: # TbCountX: #
1603, Replicator Error in SQL statement, Exception: XXX

1701, Executor Error : Initialisation failed, Exception: XXX
1702, Executor Error : Close failed, Exception: XXX
1703, Executor Error : Table: XXX exist

1704, Executor Error : Table: XXX not exist
1705, Executor Error : User XXX level ROOT is not the owner of table
1706, Executor Error : Count of field not same with structur field of table in 'iddbtbfield'
1707, Executor Error : field: XXX have fragmentation rule will not be changed
1708, Executor Error : Siteid: XXX not exist
1709, Executor Error : Siteid: XXX ever exist
1710, Executor Error : Field: XXX not found
1711, Executor Error : Table: XXX or Siteid: XXX not exist
1712, Executor Error : Siteid: XXX already exist
1713, Executor Error : While execute a command: XXX
1714, Executor Error : User: XXX, must have level # or higher !
1715, Executor Error : While load data Siteid from table 'iddbsite'

1801, OptimizerScheduler Error : Field name not found in table: XXX.XXX
1802, OptimizerScheduler Error : ORDER BY identifier must exist in SELECT argumen: XXX.XXX

Keterangan tanda :

XXX : menyatakan kata
: menyatakan angka

LAMPIRAN F

PERINTAH, OPTION, & UTILITY

F.1. PERINTAH TAMBAHAN (BUKAN STANDAR SQL):

EXTENDED TABLE CREATE *table_name* SITE *site_id* FRAGMENT *fragment_condition* [DEFAULT]

Fungsi : Menambah site-site fragmentasi table

table_name	: nama table.
fragment_condition	: aturan untuk condisi fragmentasi.
DEFAULT	: jika ada nilai yang dimasukkan tidak memenuhi aturan semua fragmentasi yang ada maka dimasukkan pada fragmentasi pada site ini.

EXTENDED TABLE DROP *table_name* SITE *site_id*

Fungsi : Menghapus fragmentasi table *table_name* pada site *site_no* tertentu.

EXTENDED SITE CREATE *site_id, 'driver', 'url', 'user', 'password'*

Fungsi : Untuk menambah site fragmentasi

EXTENDED SITE DROP *site_id*

Fungsi : Untuk menghapus site fragmentasi

COMMIT :

Fungsi : melakukan perintah commit pada transaksi yang ada.

ROLLBACK :

Fungsi : melakukan perintah rollback pada transaksi yang ada.

F.2. OPTION pada IDDBMaster.ini:

[SERVER]

OPTIMIZER = REPLICATOR / DIRECT / SEMIJOIN

REPLICATOR	: proses query menggunakan data update replicator di primary site.
DIRECT	: proses query dilakukan langsung ke tabel-tabel di DBMS lokal.
SEMIJOIN	: proses query langsung tetapi pendistribusian data dilakukan optimasi operasi semijoin.

TRANSACTION = SINGLE / CONCURRENT / NONE

SINGLE	: transaksi hanya bisa dilakukan secara satu persatu.
CONCURRENT	: transaksi dapat dilakukan secara bersamaan.
NONE	: setiap perintah dapat dilakukan tanpa memperhitungkan urutan transaksi.

INSERT = RANDOM / ORDER / FRAGMENT

RANDOM	: pemasukkan data dilakukan secara acak pada salah satu site dalam table tersebut.
ORDER	: pemasukkan data dilakukan secara urut pada salah satu site dalam table tersebut.

FRAGMENT : pemasukkan data dilakukan dengan proses pengecekan dengan aturan pada salah satu fragmen site yang sesuai dengan nilai yang masukkan ke dalam table tersebut.

[REPLICATION]

REPLICATION = SYNCHRONOUS / ASYNCHRONOUS / NONE

SYNCHRONOUS : replikasi dilakukan secara synchronous.

ASYNCHRONOUS : replikasi dilakukan secara asynchronous.

NONE : replikasi dimatikan.

TIME_ASYN_REPL = (nilai)

nilai (integer) : waktu replikasi asynchronous dalam detik.

[DBMSPRIMARY]

DRIVER : driver DBMS utama.

URL : url DBMS utama.

USER : user DBMS lokal.

PASSWORD : password DBMS utama.

F.3. OPTION pada IDDBJDBC.ini :

[LOCAL]

LOCAL = ON / OFF

ON : lakukan pemprosesan secara lokal.

OFF : lakukan pemprosesan secara global.

[DBMSLOCAL]

DRIVER : driver DBMS lokal.

URL : url DBMS lokal.

USER : user DBMS lokal.

PASSWORD : password DBMS lokal.

F.4. TOOLS Tambahan :

user (ADD|DEL|MODIFY|PRINT) *username*:

ADD : untuk menambah user.

DEL : untuk menghapus user.

MODIFY : untuk mengganti password user.

PRINT : untuk menampilkan semua user.

Level : SYSDBA | ROOT | USER | GUEST

SYSDBA : dapat melakukan semua perintah-perintah yang ada di server IDDB.

ROOT : dapat membuat/menghapus table, update, dan baca data.

USER : dapat akses update dan baca data.

GUEST : hanya akses baca data.

* Secara default IDDBMaster membuat user SYSDBA dengan password MasterKey.

LAMPIRAN G

INSTALASI & SETUP AWAL

G.1. INSTALASI

Untuk instalasi, terdapat program skrip instalasi yang dapat bekerja di windows 95 dan UNIX pada paket distribusi, (telah ditest di win95 dan Linux). Karena untuk menjalankan program Java terdapat perbedaan platform serta berbeda JVM (Java Virtual Machine) mempunyai cara berbeda untuk melihat class-class yang ada, sehingga susah untuk menyediakan semua metoda skrip penginstallaan.

Beberapa yang perlu dilakukan untuk menjalankan IDDB:

Tambahkan <InstallDir>/IDDB (atau nama directory yang mirip, seperti, <InstallDir>\IDDB (untuk win95)) ke classpath di environment variable (Sun's JDK) atau init file (seperti di Symantec Cafe) (atau java compiler /VM untuk mencari classpath). <InstallDir> merupakan directory dimana di unzip/untar/install menggunakan install.class javacc.

G.2. SETUP AWAL

Bagian ini menjelaskan mengenai menjalankan IDDB dan men-setup lingkungan IDDB, sehingga dapat menggunakan aplikasi frontend. Diasumsikan IDDB telah sukses diinstall. Beberapa langkah yang ditulis dalam seksi ini digunakan untuk semua pemakai IDDB, dan digunakan juga ke administrator site basis data. Administrator site basis data merupakan orang yang menginstall perangkat lunak tersebut, membuat direktori basisdata dan menjalankan proses. Orang tersebut tidak harus sebagai superuser dari operating sistem yang bersangkutan.

Saat IDDBMaster pertama kali dijalankan, ia tidak menemukan basis data utama (tabel dengan nama IDDBxxxxxxxx) sehingga ia akan membuatnya secara otomatis.

Yang perlu diubah pertama kali ialah pada file IDDBMaster.ini, bagian DBMSPRIMER disini dimasukkan DBMS utama yang akan menjadi data katalog dari IDDB ini.

Kemudian masukkan nama dari driver tersebut di variabel lingkungan "classpath" dari JVM (Java Virtual Machine) (contoh: classpath = d:\driver\.postgresql.jar).

Buat pemakai / user.

Jalankan IDDBMaster

Untuk menjalankan IDDBMaster ada parameter tambahan yaitu :

Jalankan : *IDDBMaster -d[1-3]*, akan ditampilkan pesan-pesan yang berguna untuk debugging troubleshooting.

Sedangkan untuk mematikan IDDBMaster, pada ketikan tombol 'Q' lalu masukkan nama user, dan jika masih ada hubungan (koneksi) dari client yang belum selesai maka akan diperingatkan apakah tetap ingin keluar, jika ya tekan 'Y' dan jika tidak tekan 'N'. Sedangkan untuk melihat status tekan tombol 'S'.

G.3. PENJELASAN CONTOH: Aplikasi Sistem Informasi Mahasiswa

Aplikasi contoh ini merupakan penjelasan penggunaan aplikasi sistem informasi mahasiswa, yang telah diterangkan pada subbab 6.3.

Sebagai contoh, basis data yang ada terpecah di tiga jurusan:

T. Industri (ti)

T.Informatika (tc)

T.Elektro (te).

File sumber di directory IDDBHome/sample/sis_info_mhsklh.

Cara mengcompile :

jalankan program *compile*.

Menjalankan program :

runti untuk jurusan T.Industri

runtc untuk jurusan T.Computer

runte untuk jurusan T.Elektro

LAMPIRAN H

PENJELASAN SINGKAT GRAMMAR JAVACC

Instruksi berikut ini untuk menunjukkan bagaimana mulai menggunakan JavaCC :

Jalankan *javacc* dengan file input grammar untuk men-generate sekelompok file Java yang merupakan implementasi parser dan lexical analyzer (atau token manager) :

javacc *file.jj*

Sekarang compile program-program Java hasil generate JavaCC :

javac *.java

Pada file grammar ini dimulai dengan bagian untuk setting semua pilihan yang diberikan oleh JavaCC. Pada bagian setting ini pilihan seringkali menggunakan nilai default. Sehingga pilihan-pilihan setting ini tidak terlalu penting. Pilihan dapat dengan lengkap digunakan atau dilewatkan saja, atau dihindari satu atau lebih dari setting pilihan secara individu. Detail dari pilihan secara individual didefinisikan dalam JavaCC documentation.

Berikut ini merupakan sebuah unit compilasi Java ditutup diantara "PARSER_BEGIN(name)" dan "PARSER_END(name)". Unit compilasi ini dapat menjadi sangat komplek. Hanya constraint di unit kompilasi yang harus didefinisikan sebuah class yang disebut "name" – sama seperti argumen PARSER_BEGIN dan PARSER_END. Nama ini yang digunakan sebagai prefix untuk file-file Java yang dibuat oleh parser generator. Code parser yang dibuat dimasukkan langsung sebelum menutup kurung dari class yang dinamakan "name".

Berikut ini daftar production. Disini terdapat dua produksi, yang mendefinisikan non-terminals "Input" dan "MatchedBraces". Dalam grammar JavaCC, non-terminal-non-terminal ditulis dan diimplementasi (oleh JavaCC) yang merupakan metode-metode Java. Ketika non-terminal digunakan disisi kiri dari sebuah production, ia digunakan untuk pendeklarasian dan sintaksnya mengikuti sintaks Java. Di bagian sisi kanan menggunakan metode yang sama dengan pemanggilan metode dalam Java.

Setiap production mendefinisikan sisi kiri non-terminal diikuti oleh sebuah karakter titik dua. Ini diikuti oleh sekelompok dari deklarasi-deklarasi dan statement-statement dengan pencabangan-pencabangan (dalam kedua hal tersebut tidak ada deklarasi, maka akan tampak seperti "{}") yang membuat seperti deklarasi umum dan statement sesuai dalam metode yang digenerate. Ini diikuti oleh sekelompok expansion yang ditutup dengan pencabangan.

Lexical token-lexical token (regular expresi) dalam sebuah grammar input JavaCC merupakan juga string-string sederhana ("{", "}", "\n", dan "\r"), atau sebuah regular expresi yang lebih komplek. Sebuah regular expresi "<EOF>" sering kali digunakan untuk mencocokkan dari akhir dari file. Semua regular expresi yang komplek ditutup dengan kurung tutup ")".

Production pertama diatas mengatakan bahwa non-terminal "Input" diperluas ke non-terminal "MethodBraces" diikuti oleh kosong atau lebih baris yang diakhiri karakter ("\n" or "\r") and lalu akhir dari file.

Production kedua mengatakan non-terminal "MatchedBraces". Memperluas token "{" diikuti oleh sebuah tambahan ekspansi bersarang dari "MatchedBraces" diikuti oleh token "}". Kurung siku-kurung siku [...] dalam sebuah file input JavaCC menyatakan bahwa ... merupakan pilihan tambahan.

JavaCC mempunyai empat lexical region :

SKIP: regular expresi yang cocok akan dilewati (dan tidak dimasukkan dalam proses parsing)

TOKEN: untuk menspesifikasikan token-token lexical

SPECIAL_TOKEN: untuk menspesifikasikan lexical token yang dilewati dalam proses parsing. dalam hal ini, SPECIAL_TOKEN mirip SKIP. Tetapi token-token tersebut dapat dikembalikan dengan aksi parser yang dihandle dengan cocok.

MORE: untuk menspesifikasikan token secara terpisah. Token secara lengkap dibuat dalam urutan dari beberapa MORE diikuti oleh sebuah TOKEN atau SPECIAL_TOKEN.

Expresi regular [...] boleh juga ditulis sebagai (...)? , hal ini merupakan dua bentuk yang sama

Konstruksi lain yang sering tampil dalam expresi regular adalah:

(e)+ : kehadiran e satu atau lebih.

(e)? : kehadiran dari e boleh ada atau tidak.

(r1 | r2 | ...) : salah satu dari r1, r2, ...

Catatan : ini dapat dilakukan perintah bersarang (nested) satu sama lainnya, jadi boleh mempunyai seperti

((e1 | e2)* [e3]) | e4

Karakter-karakter yang dispesifikasikan dalam ... . karakter-karakter ini dapat sebuah karakter atau karakter dalam jangka tertentu. Sebuah "~" sebelum konstruksi ini merupakan sebuah pattern yang cocok oleh beberapa character yang tidak di spesifikasikan oleh ... .

["a"-"z"] cocok dengan semua huruf kecil

~[] cocok dengan semua karakter

~["\n","\r"] cocok dengan beberapa karakter kecuali dengan baris baru.

Ketika sebuah ekspresi regular digunakan, ia menggunakan sebuah nilai dari sebuah tipe "Token". Ini dibuat dalam directory parser sebagai "Token.java". dalam contoh diatas didefinisikan sebuah variable dari tipe "Token" dan memberikan nilai dari regular expresi ke dalamnya